ActRec - A Non-monotonic Reasoning Tool for Activity Recognition

Saeed Partonia

January 25, 2015
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Juan Carlos Nieves Sánchez
Examiner: Frank Drewes

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Human activities usually have a motive and are driven by goal directed sequence of actions. Recognizing and supporting human activities is an important challenge for ambient assisted living. Human activity recognition has a wide scope of application areas, e.g., aged care support, health care, smart homes, natural disasters and energy efficient urban spaces. Different techniques have successfully been applied to infer human activity, including machine learning and data mining. These data driven techniques work well within a particular domain and situations in which they are initially set in. However two main drawbacks with such methods have been observed in the literature: they are domain dependent and also require large amount of data annotation for model training. Hence, different authors have argued for exploring complex activity recognition techniques that not only rely on data but also involve domain knowledge is necessary.

Against to this background, in this project, we explore non-monotonic reasoning technics in order to capture domain knowledge in terms of action specification languages. By considering an action specification language, called C_TAID, and Answer Set Programming, we propose and develop a system called ActRec system which takes background information into consideration and operates independently from the environmental factors. We also explore a novel definition of activity which is used in the implementation of ActRec.
# Contents

1 Introduction 1  
1.1 Problem Statement 1  
1.2 Objective 2  
1.3 State of the art 3  
1.4 Thesis Outline 5  

2 Background 7  
2.1 Answer Set Programming 7  
2.2 Action Languages 9  
2.2.1 Action Language $\mathcal{CTAID}$ 9  
2.3 ASP Solvers 12  

3 Theoretical Framework 15  
3.1 Activity Definition 15  
3.2 Activity Recognition Algorithm 16  
3.2.1 Activity Selection 18  

4 Description of ActRec System 21  
4.1 System requirements 22  
4.2 System Parts 22  
4.2.1 Mapper 22  
4.2.2 Solver 24  
4.2.3 Recognizer 24  
4.3 Data Flow 24  
4.3.1 Data Input 24  
4.4 The Code 26  

5 Evaluation 29  
5.1 Experiments 29  
5.1.1 Formulating the environment 29  
5.2 Discussion and Interpretation of Data 31  
5.3 Future Work 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusion</td>
<td>35</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>37</td>
</tr>
<tr>
<td>References</td>
<td>39</td>
</tr>
<tr>
<td>Source Code</td>
<td>43</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 A representation of an atomic activity. ............................................. 2  
1.2 A complex (concurrent) activity. .................................................. 2  
1.3 Overall Diagram of thesis. ............................................................... 3  

2.1 ASP solving order. ........................................................................... 13  

3.1 A trajectory. ...................................................................................... 17  
3.2 A recognized activity using initial state (green) and final state(red). .... 17  
3.3 An ongoing activity with initial state (green). .................................... 17  
3.4 Activity recognized by a sequence of actions \([a_0, a_1, a_2]\). .................. 18  

4.1 Work flow of ActRec System. ............................................................. 21  
4.2 Different components of the Mapper module. .................................... 22  
4.3 The logical view of the ActRec database. ......................................... 23  
4.4 Adding fluents to the system. ............................................................ 25  
4.5 Defining rules and constraints to the system. .................................... 26  
4.6 Adding a desired activity to the system to check for recognition. ......... 26  
4.7 Data flow of the ActRec system. ....................................................... 26  
4.8 Terminal output for a recognized activity with two states. ................. 27  

5.1 Activities grouped in different categories. ....................................... 32  
5.2 Activities repeating over time and forming a pattern. ......................... 32
List of Tables

2.1 ASP Grounders .................................................. 14
2.2 ASP Solvers .................................................. 14

5.1 Fluents and actions used in each scenario .................. 30
5.2 Results of running ActRec algorithm on two scenarios .......... 31
Chapter 1

Introduction

Human activities usually have a motive and are driven by goal directed sequence of actions [1]. Recognizing and supporting human activities is an important challenge for ambient assisted living. Indeed, human activity recognition has a wide scope of application areas, e.g., aged care support, health care, smart homes, natural disasters and energy efficient urban spaces [2, 3, 4]. Different techniques have successfully been applied to infer human activity, including machine learning and data mining [5]. These data driven techniques work well within a particular domain and situations in which they are initially set in. However, there are some drawbacks with these techniques. First, the activity recognition model needs to be changed once the environment is changed. Additionally they suffer from large amount of data collection and annotation. Hence, building complex activity recognition techniques that not only rely on data but also involve domain knowledge are necessary [5].

In this project, our aim is to define and implement an open source system which takes the background information into consideration but also operates independently from the environmental factors. To this end, we explore non-monotonic reasoning theories for activity recognition. We take advantage of Answer Set Programming (ASP) in the context of Action Specification Languages to discover not only answers to whether an activity is occurred, but also an explanation to the performed activity.

1.1 Problem Statement

Different techniques have successfully been applied to infer human activities from a given environment, including machine learning [2, 6] and data mining combined with the use of semantics [7]. Machine learning methods are among most applied techniques. For instance, in our previous work, we have developed a data driven system with vision and machine learning techniques as a part of AS-A-PAL project at Umea University, Sweden [8]. In this setting, we observed that data-driven approaches works well within a particular domain and situation in which they are initially set in. However, once the environment is changed, for instance, the activity recognition model needs to be retrained [9, 10]. Another drawback with data-driven techniques is they suffer from large amount of data collection and annotation for training [2, 11, 12]. Additionally, with existing data driven techniques, interleaved and concurrent activities are not accurately recognized, because it is vary time consuming to train activity models for every possible way in which activities are in interleaved or performed concurrently [2, 11, 12]. Hence, there is a need to build complex activity recognition techniques that not only rely on data but also involve domain knowledge. To infer complex
activities, the authors in [5] propose reasoning about context and atomic activities. An atomic activity \( A \) is defined as a unit level activity which cannot be broken down further given application semantics. Figure 1.1 depicts an atomic activity, while Figure 1.2 shows a complex one.

Action Languages [13] usually employed for reasoning about the behavior of the agents, e.g., human activities. They are capable of compactly describing dynamic domains and usually implemented through compilation to Answer Set Programming (ASP). Moreover, they also give an explanation rather than a firm probability or yes/no as a solution to a problem. Some authors have applied action languages for modeling behavior of biological networks via answer set programming [14]. To this end, they proposed an action language called \( C_{TAID} \). Biological systems are similar in their behavior to the human’s in some aspects. First, they both may have simple or complex activities. The activity of ‘walking’, for instance, may be regarded as simple activity however, ‘Shopping’ activity could be considered as complex one which also includes ‘walking’. Secondly, human behavior is similar to biological systems in a sense that human behavior can be described by causal rules. In other words, any activity must meet a certain preconditions to be happen, and there is always a reason why an activity is performed. For example, in order to ‘prepare coffee’ a certain requirements such as ‘boiling water’ are necessary to met. Therefore, \( C_{TAID} \) seems to be a suitable language for modeling human activity.

Figure 1.1: A representation of an atomic activity.

Figure 1.2: A complex (concurrent) activity.

In our approach, we introduce a novel definition for an activity by considering concepts from [5] in terms of action specifications. We also present an algorithm for inferring activities by considering the suggested definition of an activity. Moreover, an open source software is implemented and tested with some scenarios based on the proposed ideas.

1.2 Objective

In this project, the goal is to implement a meta-interpreter of declarative action specifications [1] in order to answer questions about human activities such as:

- Which activity was performed?
1.3 State of the art

How was the activity performed?
How can we be sure that the activity was performed?

In order to answer these questions, we will explore a potential definition of an activity. Once we defined what is an activity, a meta-interpreter, which takes as inputs observations from a world\(^1\) and action specifications of the world\(^2\) (Figure 1.3), will be implemented. This meta-interpreter will map its inputs into an Answer Set Program. The answer sets of this program will be explored by a new algorithm to find out whether a specific activity is recognized or not. The whole task is summarized as follows:

- To explore a definition of an activity.
- Implementing a meta-interpreter to feed action specifications and observations to an ASP-solver (Figure 1.3).
- Implementing an activity recognition algorithm with respect to the activity definition.
- Evaluation of the implemented system.

![Figure 1.3: Overall Diagram of thesis.](image)

1.3 State of the art

Human activity recognition aims to recognize the actions and goals of one or several agents on the agent’s actions and the environmental conditions. Due to its strength in providing

\(^1\)This world depends on the applications domain. For instance, it can be a smart environment. In this setting, the observations from the world can be readings from sensors.

\(^2\)An action specification basically is a declarative specification of world in terms of fluents and actions [1].
personalized support for many different applications and its connection to many different fields of study such as human-computer interaction, sociology and many others, it has captured the attention of researchers in computer science. Human activity recognition has a wide scope of applications in areas, e.g., aged care, health care, smart homes, natural disasters and energy efficient urban spaces [2, 3, 4]. It can also be applied to the field of personal informatics, by helping humans better understand and analyze their behavior [15]. Consider the following example to understand the importance of the human activity recognition from: Henrik is a widower living alone in his studio apartment. Making breakfast is a daily activity for Henrik. During the breakfast scenario, he interacts with several devices in the kitchen. He lights the stove to boil some eggs, switched the toaster on, turns the coffee machine on and so on. After he has finished breakfast, a computer generated voice gently reminds him to turn off the stove. Later that day, his daughter using an application installed in her smartphone scans a check-list, which was created by a sensor network in her father’s apartment. She finds that her father is eating normally, taking his medicine on schedule, and continuing to manage his daily life on his own.

Cooking, eating, walking and house work are some examples of Activities of Daily Living (ADLs). Each ADL has more than one sub-activity. Cooking, for instance, may involve ‘lighting the stove’, ‘putting a pan on stove’, ‘opening the fridge’, ‘taking some ingredients from fridge’, ‘putting the ingredients in the pan’ and so on. One can perform these activities in different ways and different times. Hence there is no precise definition of what is an activity. However, some authors suggest these intuitive definitions; for instance, according to [5], these sub-activities are called atomic activities because they cannot be broken down further. Atomic activities belonging to different ADLs may interleave or occur concurrently [6]. A complex activity consist of a number of atomic activities. Complex activities may also interleave or occur concurrently. ‘Cooking an Omelet’ can be considered as a complex activity since this activity is composed by atomic activities, e.g., ‘walking into kitchen’, ‘picking a pan’, ‘standing close to stove’ and so on. Additionally one can perform these activities in different ways and orders. Various techniques have been applied to infer human activities from a given environment. Machine learning [2, 6] and data mining are among most applied techniques [7]. In this setting, it has been observed that these techniques work well within a particular domain and situation in which they are initially set in. However once the environment and the infrastructure is changed, the activity recognition model needs to be retrained [9, 10]. Additionally they suffer from large amount of data collection and annotation for training [2, 11, 12]. Another major problem with existing data driven techniques is that interleaved and concurrent ADLs are not accurately recognized. Probabilistic models such as naive Bayes, decision trees, Hidden Markov Models (HMM) and dynamic Bayesian Networks (DBN) are also applied for the problem of activity recognition. It turns out that probabilistic models work well in a workshop or industry settings [17, 18, 19]. In this setting, activities are performed in the same sequence each time. However, accuracy drops when interleaved or concurrent activities occur [2, 12]. For daily life activity recognition, data mining techniques show promising results [20]. While they perform well in interleaved and concurrent activities in the mentioned environments, issues such as variations in duration of activities adversely affect the system [21]. To this end, a complex activity recognition technique is required that not only rely on data but also involve domain knowledge, too. The authors in [21] attempt to use Temporal Analysis for recognizing concurrent tasks. Such models also require collection and labeling of large amount of data and do not involve the background knowledge. In [22], the authors use ontological constructs and reasoning to recognize complex activities. However their work does not attempt to recognize concurrent

\footnote{The example is adapted from an article at http://en.wikipedia.org/wiki/Action_language.}
and interleaved activities. A recent work proposed and developed a Context Driven Activity Theory (CDAT) which also involves background information [5]. According to this work, a complex activity can vary each time a user performs it. It also can have different start, end and life span (duration). An interleaving activity can be represented as \((CA_3 | CA_4 | CA_6)\) for instance, where each \(CA_i\), \(i \in \{3, 4, 6\}\) represents a complex activity and ‘\(|\)’ denotes the concurrency. To infer complex activities, authors in [5] propose reasoning about context and atomic activities. An atomic activity \(A\) is defined as a unit level activity which cannot be broken down further given application semantics. Here, application semantics refer to apply specific conditions and constraints. Likewise, a context attribute \(C\) is defined as any piece of data or information at time \(t\) that is used to infer an activity or a situation. In [23], a framework based on formal languages and reasoning to solve activity recognition problem is proposed. This approach is based on the concept of so called ‘intended actions’. In this context, agent intends to perform an action. If the agent is not able to execute the intended action at a specified time, by definition, intentions persist until the agent successfully execute the action. In our approach, we explore atomic and complex activities from the point of view of [5]. Additionally, we propose a novel definition for the activity.

### 1.4 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 focuses on the background knowledge required to understand the suggested approach. This chapter consists of an introduction to Answer Set Programming and Action Languages. Chapter 3 explains a novel definition of an activity. Moreover, an algorithm for activity recognition with respect to the proposed definition and considering action specifications is discussed. Chapter 4 gives a description on the implemented system. In Chapter 5, we will discuss the results of the implemented system, followed by a discussion of future work. Finally, Chapter 6 is dedicated to the conclusion of the thesis.
Chapter 2

Background

Action Languages are capable of effectively describing dynamic domains. Their operational semantic is usually implemented by compilation to Answer Set Programming (ASP) or Satisfiability Checking (SAT) [24]. Both approaches are capable of solving problems with millions of variables. Nevertheless, ASP has some advantages over SAT [14]. First, ASP is more expressive than SAT. Additionally, ASP, due to its root in knowledge representation, has a richer input language. Therefore, many tools prefer ASP over SAT for implementing action languages. Potassco [25] a project from the University of Potsdam, Germany, which acts as an umbrella for variety of tools for action languages, use ASP for implementation. In this thesis we build our system on top of some of existing tools in Potassco project. Hence, we dedicate the next section to give a short introduction to ASP programming. The rest of the chapter introduces the basic concepts of action languages and tools for solving them.

2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative programming approach that aims to solve NP-hard search problems. The basic idea of declarative programming is to represent the given problem by a set of rules, find answer sets for the program using a solver, and finally extract the solutions from the answer sets. The rest of this section is adopted from the original paper on ASP by Gelfond [26].

Programs in ASP consist of the rules of the form

\[ A_0 \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \]  \hspace{1cm} (2.1)

where \( A_i \) is a propositional atom and \( \text{not } A_i \), \( (1 \leq i \leq n) \) shows there here is no evidence that we hold \( A_i \). Intuitively, this expression means that if atoms \( A_1, \ldots, A_m \) hold and \( A_{m+1}, \ldots, A_n \) do not hold, then we may derive \( A_0 \) holds. We call \( A_0 \) as head and \( A_1, \ldots, A_m \) as body of the rule. There are some special cases for a rule. For instance, a rule is called a fact if the body of the rule is dropped, i.e

\[ A_0 \leftarrow \]  \hspace{1cm} (2.2)

\( A_0 \) is considered as a fact. Likewise, if the head of the rule is missing, it is called a constraint or a goal. The rule

\[ \leftarrow A_1, \ldots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \]  \hspace{1cm} (2.3)

is called a constraint. A logic program is defined subsequently.
Chapter 2. Background

Definition 1: A logic program is defined as a pair \( \langle \sigma, \Pi \rangle \), where \( \sigma \) is a signature, and \( \Pi \) represents a set of logic programming rules.

Such pair is often denoted by its second element \( \Pi \) and \( \sigma(\Pi) \) represents the corresponding signature. A signature contains the symbols for describing a language. For a logic program \( \Pi \), the answer set semantics assigns a collection of answer sets to program \( \Pi \). We also define state as a set of literals over signature \( \sigma \) which are true under the interpretation of their symbols. Let \( X \) be a state of \( \sigma(\Pi) \). We say that \( X \) is closed under \( \Pi \) if, for every rule of the form \( \text{head} \leftarrow \text{body} \) of \( \Pi \), \( \text{head} \) is true in \( X \) whenever \( \text{body} \) is true in \( X \).

Definition 2 (answer set - part 1): A state \( X \) of \( \sigma(\Pi) \) is an answer set for \( \Pi \) if \( X \) is minimal, in the sense of set-theoretic inclusion, among the sets closed under \( \Pi \).

In order to extend this definition, suppose for a program \( \Pi \), let \( X \) be a state of \( \sigma(\Pi) \). the reduct, \( \Pi^X \), of \( \Pi \) relative to \( X \) is the set of rules

\[ l_0 \leftarrow l_1, \ldots, l_m \]

for all rules 2.1 in \( \Pi \) such that \( l_{m+1}, \ldots, l_n \notin X \). Thus \( \Pi^X \) is a program without default negation.

Definition 3 (answer set - part 2): A state \( X \) of \( \sigma(\Pi) \) is an answer set for \( \Pi \) if \( X \) is an answer set for \( \Pi^X \).

Gelfond and Lifchitz [13] introduce a method to obtain reduct of a program \( \Pi \), for a given state \( X \) by

- removing rules with \( \text{not} \ a \) in the body for each \( a \in X \).
- removing literals \( \text{not} \ a \) from all other rules.

When the reduct of a program is obtained, one can check if \( X \) is an answer set of \( \Pi \) by ensuring if \( X \) is a minimal model of \( \Pi^X \) with respect to set inclusion \((\subseteq)\).

Definition 4 (entailment): A program \( \Pi \) entails a literal \( l(\Pi \models l) \) if \( l \) belongs to all answer sets of \( \Pi \).

Consider a logic program, for instance: \( \Pi = \{ a \leftarrow \text{not} \ b. \}
\]

For this program, possible states that are close under \( \Pi \) are:

- \{a\}
- \{b\}
- \{a, b\}

If we determine the reduct for the first state, \( X = \{a\} \), we end up with \( \Pi^X = \{a\} \). Since in this case, \( X \) is the minimal set closed under \( \Pi^X \), then it is an answer set for \( \Pi \). The same is true for \( X = \{b\} \). For the state \( X = \{a, b\} \), we have \( \Pi^X \) as follows:

- remove \( a \) from \( \Pi^X \) since \( \text{not} \ a \) is included in one of the rules.
- remove \( b \) from \( \Pi^X \) since \( \text{not} \ b \) is included in one of the rules.

Thus, we end up with \( \Pi^X = \{\} \). Since \( X = \{a, b\} \not\subseteq \{\} \) then \( X = \{a, b\} \) is not an answer set for \( \Pi \).
2.2 Action Languages

Action languages were designed to specially describe dynamic domains. Gelfond and Lifschitz [13] first introduced the concept of action languages in 1990s. Language \( A \) is considered as the most basic action language and many languages like \( A^0 \) extend \( A \). \( B[13] \), \( C[13] \) and \( K[13] \) are among action languages that are more expressive than \( A[13] \). \( C_{T,AID} \) [14] which is built on \( A^0 \) and \( C \) is another action language originally introduced for describing biological networks. In addition to inheriting features like trigger and inhibition rule from its predecessors, \( C_{T,AID} \) adds a feature called allowance rule. These capabilities makes \( C_{T,AID} \) a suitable choice for describing human activity. The rest of this section is adopted from the original work by [14].

2.2.1 Action Language \( C_{T,AID} \)

The alphabet of the language \( C_{T,AID} \) consists of two disjoint and non empty sets of symbols; 1) a set of fluents \( F \) and 2) a set of action names \( A \). Essentially, a fluent literal is behaved as a propositional value and is either true or false and possibly preceded by a \( \neg \). For instance, in a scenario of ‘Making Pasta’, a simple fluent could be as simple as

\[
\text{vessel\_heat\_on}
\]

or

\[
\text{vessel\_water\_full}
\]

Informally, a fluent expresses the property of an object in a world. Consequently, actions can influence fluents. A simple action for ‘making pasta’ scenario could be

\[
\text{pasta\_sauce\_add}
\]

A collection of fluents is called a State and denoted by \( S \). We call fluent \( f \) holds in state \( S \) if \( f \in S \). Likewise, we say a fluent literal \( \neg f \) holds in \( S \) if \( f \notin S \).

\( C_{T,AID} \) consists of three sub-languages: 1) Action description language which expresses the general knowledge about the system. 2) Action observation language which is used to describe a distinct point of time in the system. And 3) Action query language for reasoning about the system. This section gives an overview of the action language \( C_{T,AID} \).

Action Description Language

We start introducing the syntax of action specification languages by some definitions.

**Definition 5:** A collection of the rules of the following form define a domain description \( D(A,F) \) in \( C_{T,AID} \):

- A dynamic causal rule, states that \( f_1, \ldots, f_n \) holds after action \( a \) occurs, with preconditions \( g_1, \ldots, g_m \) is an expression of the form:

\[
(a \text{ causes } f_1, \ldots, f_n \text{ if } g_1, \ldots, g_m)
\]  \hspace{1cm} (2.4)

The if part can be dropped if there are no preconditions of the form \( g_1, \ldots, g_m \).
Chapter 2. Background

- **Static causal rule**, describes immediate dependencies between fluents and is defined as:
  \[ (f_1, \cdots, f_n \text{ if } g_1, \cdots, g_m) \]  
  (2.5)

- **Triggering rule**, states that if preconditions \( f_1, \cdots, f_n \) hold, action \( a \) occurs immediately and is of the form:
  \[ (f_1, \cdots, f_n \text{ triggers } a) \]  
  (2.6)

- **Allowance rule**, states that if preconditions \( f_1, \cdots, f_n \) holds, action \( a \) can occur but not necessarily.
  \[ (f_1, \cdots, f_n \text{ allows } a) \]  
  (2.7)

- **Inhibition rule** states that if preconditions \( f_1, \cdots, f_n \) hold, action \( a \) cannot occur.
  \[ (f_1, \cdots, f_n \text{ inhibits } a) \]  
  (2.8)

- A **non-concurrency rule** of the form:
  \[ (\text{nonconcurrency } a_1, \cdots, a_n) \]  
  (2.9)

  is a constraint that states actions \( a_1, \cdots, a_n \) cannot occur at the same time.

- Finally, a rule of the form
  \[ (\text{default } f) \]  
  (2.10)

  is used to assign a default value for fluent \( f \).

One can imagine a causal rule for making pasta scenario as:

\( \text{pastaboil causes pastacooked if vesselonheat, vesselwaterfull, vesselpastafull} \)

As mentioned earlier, a state is a set of fluents that hold in that particular set. In order to define what is a state in \( CTAID \) specification, first we need to give a definition to Interpretation:

**Definition 6**: Given a domain description \( D(A, F) \), we say that \( I \) is an interpretation over \( F \) if it is a consistent and complete set of fluents.

Now that we defined interpretation, we can introduce a mathematical representation for state as given below.

**Definition 7**: A state \( s \) is defined as an interpretation over \( F \) if the following holds: for every causal rule, i.e., \( (f_1, \cdots, f_n \text{ if } g_1, \cdots, g_m) \), we have \( \{f_1, \cdots, f_n\} \subseteq s \) if \( \{g_1, \cdots, g_m\} \subseteq s \).

Taking our toy example of making pasta, a state in this settings could be a set of fluents representing a stage in a cooking process, for example:

\[ \text{state}_0 = \text{heat\_on, vessel\_on\_heat, vessel\_water\_full} \]
\[ \text{state}_1 = \text{vessel\_has\_boiled\_water, vessel\_pasta\_full} \]
In this example, \(state_0\) states that

\[
\begin{align*}
heat \rightarrow on, \ & vessel \rightarrow on \rightarrow heat, \ & vessel \rightarrow water \rightarrow full
\end{align*}
\]

must hold in that particular state. Similarly,

\[
\begin{align*}
vessel \rightarrow has \rightarrow boiled \rightarrow water, \ & vessel \rightarrow pasta \rightarrow full
\end{align*}
\]

also need to hold in State 1. Now that we have our states defined, we need a way to define \textit{transition} between those states. Thus we define a transition as follows.

**Definition 8:** Let \(S\) be a set of states of domain description \(D(A, F)\) and \(B \subseteq A\) be a set of actions in \(s \in S\). The resulting state \(s'\) is determined by \textit{transition relation} \(\Phi \subseteq S \times 2^A \times S\) after all actions in \(B\) is executed.

It is possible that for an action can occur or cannot occur on a given state \(s\). The notion of \textit{applicable, active} and \textit{passive} rules are defined as follows.

**Definition 9:** Let \(s\) be a state of a given domain description \(D(A, F)\).

- An inhibition rule \((f_1, \cdots, f_n \text{ inhibits } a)\) is active in \(s\) if \(f_1, \cdots, f_n\) hold in \(s\). Otherwise it is passive.
- A triggering rule \((f_1, \cdots, f_n \text{ triggers } a)\) is active in \(s\) if \(f_1, \cdots, f_n\) hold in \(s\) and all inhibition rules of \(a\) is passive. Otherwise is called passive in \(s\).
- An allowance rule \((f_1, \cdots, f_n \text{ allows } a)\) is active in \(s\) if \(f_1, \cdots, f_n\) hold in \(s\) and all inhibition rules of \(a\) is passive. Otherwise is called passive in \(s\).
- A dynamic causal rule \((a \text{ causes } f_1, \cdots, f_n \text{ if } g_1, \cdots, g_n)\) is applicable in \(s\) if \(g_1, \cdots, g_n\) hold in \(s\).
- Likewise, a static causal rule \((f_1, \cdots, f_n \text{ if } g_1, \cdots, g_n)\) is applicable if \(g_1, \cdots, g_n\) hold in \(s\).

Transition system defines a way to traverse between different states. Basically, different states are chained together by a transition system. The combination of a several states and a transition system could be formulated as a \textit{trajectory}.

**Definition 10:** A trajectory of the form \(\langle s_0, A_1, s_1, \cdots, A_n, s_n \rangle\) of a domain description \(D(A, F)\) is defined as a sequence of states \(s_i\) and sets of actions \(A_i\) for \((0 \leq i \leq n)\), and expresses the possible evolution of the system with respect to \(D(A, F)\). A trajectory ensures that the following conditions hold.

1. There is a reason why an action occurs (or why does not occur)
2. Actions belonging to all active triggering rules in \(s_i\) is included in \(A_{i+1}\)
3. No actions for which all triggering rules are passive in \(s_i\) is included in \(A_{i+1}\)
4. No actions for which all allowance rules are passive in \(s_i\) is included in \(A_{i+1}\)
5. No inhibited actions are included in the set of actions.
Action Observation Language

The general knowledge about the environment is described by the action description language. Action observation language then comes handy to describe particular states and occurrence of actions at a distinct point of a time. Basically, it describes the state of the system with the following expressions.

An expression of the form:

\[(f \text{ at } t_i)\] (2.11)

is used to express the value of the fluent \(f\) at time \(t_i\) is true. Fluent \(f\) is preceded by a \(\neg\) if it has the value \(false\) at time \(t_i\). In our toy example, \(heat - on \text{ at } t_0\) for instance represents the value of fluent

\(heat\_on\)

as \textbf{true} at time 0.

Additionally, an expression of the form:

\[(a \text{ occurs-at } t_i)\] (2.12)

states that action \(a\) occurs at time \(t_i\). Similarly, action \(a\) is preceded by a \(\neg\) says that \(a\) does not occur at time \(t_i\).

Action Query Language

Action query language is applied to reasoning about the system. Basically, a query is defined as:

\[(f_1, \cdots, f_n \text{ after } A_1 \text{ occurs-at } t_1, \cdots A_m \text{ occurs-at } t_m)\] (2.13)

where \(f_1, \cdots, f_n\) denote fluents, \(A_1, \cdots, A_m\) and \(t_1, \cdots t_m\) are sets of actions and time points respectively.

2.3 ASP Solvers

ASP, is theoretically based on the answer set semantics of logic programming introduced in section 2.2.1. Basically, in ASP, search problems are reduced to computing answer sets or stable models. Then, a program called answer set solver, is employed to perform search. A ASP program consist of rules of the form

\[<\text{head}>: -<\text{body}>.\]

A \textit{fact}, the rule with empty body is presented as:

\textit{cloudy}.

A program can also include a \textit{choice} rule of the form:

\[\{s,t\}: -p.\]
Which literally says that if \( p \) is included in the answer sets, then arbitrarily choose one of the atoms \( s, t \) to be included. It also allows for constrained choice rules. For example, a rule of the form:

\[
1\{p, q, r\}.
\]

means that choose at least one of the atoms \( p, q, r \) but not more than two. Constraints are added to programs to eliminate some its answer sets. A rule such as:

\[
:\neg s, \not\! t
\]

assures that \( s \) is generated only if \( t \) is not generated.

Figure 2.1: ASP solving order.

Basically, ASP solving process works in two stages: **Grounding** and **Solving**. Current answer set solvers work on variable-free programs, therefore, a program needs to be grounded before it passed to the ASP solver (Figure 2.1). By grounding, a program with variables is replaced by an equivalent one without variables. Table 2.1 represents information on the three grounders currently most broadly used in ASP. Among those, *gringo* [25] is recognized as the most standard grounder by the ASP community [27]. After grounding, a simple program like:

\[
\begin{align*}
p(a) & . \ p(b) . \ p(c) . \\
q(X) & : = p(X) . \\
2 & \{ r(X) : p(X) \}.
\end{align*}
\]

turns into:

\[
\begin{align*}
p(a) & . \ p(b) . \ p(c) . \\
q(a) & : = p(a) . \\
q(b) & : = p(b) . \\
q(c) & : = p(c) . \\
2 & \{ r(a), r(b), r(c) \}.
\end{align*}
\]

After the program is grounded, it is ready to be solved by a solver. Table 2.2 shows some of the current off-the-shelf ASP solvers. Again, among those, solver *Clasp* [25] is considered as the most popular solver by the ASP community and it won several SAT solver competitions.

Typically, the output of grounder is piped to a solver. Thus the invocation scheme of a ASP program is as follows:

\[>>\text{gringo [ options | files ] | clasp [ options | number]}\]

This mechanism could be replaced by calling *clingo* instead of gringo and clasp. *Clingo* [25] is another tool from Potassco project [25] that combines both gringo and clasp into a uniform system. So the previous command could be presented as its equivalent by invoking clingo as:
Grounder

<table>
<thead>
<tr>
<th>Solver</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPARSE</td>
<td><a href="http://www.tcs.hut.fi/Software/smodels/">www.tcs.hut.fi/Software/smodels/</a></td>
</tr>
<tr>
<td>DLV</td>
<td><a href="http://www.dbai.tuwien.ac.at/proj/dlv/">www.dbai.tuwien.ac.at/proj/dlv/</a></td>
</tr>
<tr>
<td>GRINGO</td>
<td>potassco.sourceforge.net/#gringo/</td>
</tr>
</tbody>
</table>

Table 2.1: ASP Grounders.

>> clingo [ options | files | number]

The ‘number’ argument indicates the maximum number of answer sets to be computed. By default, if exists, only one answer set is computed. However, passing 0 as number argument means computing all answer sets.
Chapter 3

Theoretical Framework

In this chapter, we give a novel definition to an activity along with other definitions required to define activity. We present the algorithm which is used in the ActRec system for activity recognition. The ActRec algorithm accepts answer sets of the ASP program in the form of trajectory, defined in Section 3.2, and outputs a value representing the probability of the (recognized) activities. A complexity analysis of the algorithm is also presented following the algorithm description. The ActRec system is described in more details in Chapter 4.

3.1 Activity Definition

In this section we propose a novel definition to activity. Our proposed definition is based on concepts of action specification languages [13] as described in Chapter 2. In a recent work [5], authors proposed a Context Driven Activity Theory (CDAT) and an algorithm to recognize complex activity. They introduce the concept of ‘context attribute’ which is defined as any type of data that is used to infer an activity at time $t$ and is used to capture the background information. They also propose the algorithm CARALGO for activity recognition. Our approach differs from the work described above in several aspects. First, we use ‘fluent’ in order to capture the background information instead of context attribute. Unlike context attribute, fluents are propositional logic predicates, i.e. true or false rather than a continuous threshold values. In this case, an action is executable if a certain set of preconditions are held. Secondly, our proposed algorithm is capable of recognizing a complex activity given a set of atomic or basic activities. The algorithm also can take the order of atomic activities into account when recognizing a complex activity, if necessary. To this end, we dedicate the rest of this section for some definitions which are prerequisite to define the concept of activity.

A ‘basic activity’ is defined as follows.

**Definition 11:** Let $S$ be a set of fluents. A basic activity is of the form $<S_I, S_F>$ such that $S_I \subseteq S$, $S_F \subseteq S$ and $S_I \neq S_F$. $S_I$ and $S_F$ are called initial and final states, respectively.

An activity may involve one or several actions. These actions may follow a certain order or not. For example, in "Breakfast scenario", the order of performing actions, e.g. eating cereal, drinking, etc. may not be so important. However, for some activities, it is necessary that the actions must be performed in a certain order. For instance, for an activity of "Ordering pizza", one usually picks up the phone, dials the restaurant’s number, orders
pizza, pays the order and finally receives pizza. Apparently it is not possible to receive pizza before ordering it. Hence, it is necessary to differentiate between two activities by defining A-activity and O-activity for non-ordered and ordered actions, respectively.

**Definition 12:** Let $A$ be a set of Actions. An A-activity is of the form $<S_I, A, S_F>$ such that

- $<S_I, S_F>$ is a basic activity.
- $\exists$ a trajectory $T = [S_0a_1S_1 \cdots a_nS_n]$, such that $S_0 = S_I, S_F = S_n$ and $a_i \in A, (1 \leq i \leq n)$.

**Definition 13:** Let $A$ be a set of Actions. An O-activity is of the form $<S_I, A, L_A, S_F>$ such that

- $<S_I, S_F>$ is a basic activity.
- $\exists$ a trajectory $T = [S_0a_1S_1 \cdots a_nS_n]$ such that $S_0 = S_I, S_F = S_n$ and $a_i \in A, (1 \leq i \leq n)$.
- $L_A = [a^1, a^2, \cdots , a^m]$ is an ordered set, by taking into account an order in a set of actions.

$T_A$ is a function which returns the set of actions which appear in $T$. $L_a$ is a function which returns the set of actions which appear in $L_A$. Let $a_i, a_j \in T_A (1 \leq i, j \leq n)$, $a^p, a^q \in L_a, (1 \leq p, q \leq m)$ and $a_i = a^p$ and $a_j = a^q$ then

$$i < j \text{ if } p < q.$$

### 3.2 Activity Recognition Algorithm

The ‘ActRec’ algorithm is used to recognize and infer complex activities with the help of fluents and actions. Before we start describing the algorithm, it is worth mentioning that the answer sets are obtained from the ASP solver are mapped through a function into the form of a trajectory defined in Definition 10.

**ActRec Algorithm**

ActRec Algorithm is shown as Algorithm 3.2 and works as follows. An activity could be recognized in two ways. One way is to provide two sets of fluents to the algorithm; an initial state and a final state. The initial state is a set of fluents that represent the preconditions for an activity to be occurred. For instance, in the ‘Making Pasta’ scenario, a simple initial set might be as

$$\text{initial state} = \{\text{heat on, vessel on heat, vessel water full}\}$$

likewise, the final state is a set of fluents showing values of the fluents which are necessary to be satisfied when an activity is performed. For instance a final state for the making pasta scenario can be:

$$\text{final state} = \{\text{pasta sauce mixed, pasta cooked}\}$$
3.2. Activity Recognition Algorithm

In our case, starting from the state 0 in the trajectory, the algorithm looks for a state in which all the fluents of the initial state are satisfied. If such state is found, it marks the state as 'initial state'. Otherwise, the search process is terminated with a failure message. Once the initial state is found, the search process continues the next state in the trajectory. In this time, the algorithm looks for a state that is a super set of the final state, i.e., all the final state's fluents hold in the state. Once again, if such state is found, the algorithm returns a success message along with the initial and final states. Otherwise it indicated that this is an ongoing activity meaning that it has been started but not finished yet. Figures 3.1, 3.3 and 3.2 show a trajectory, an ongoing activity and a recognized activity, respectively.

![Figure 3.1: A trajectory.](image)

![Figure 3.2: A recognized activity using initial state (green) and final state(red).](image)

![Figure 3.3: An ongoing activity with initial state (green).](image)

The second possibility to infer an activity from a trajectory is to check for an occurrence of a set of actions. Consider our ‘making pasta’ activity once again. In order to make a pasta dish a certain steps are necessary to be done. For instance, we can imagine that a
sequence of actions like

\[ \text{action list} = [\text{vessel water boil, pasta boil, pasta cooked, drain, pasta sauce add}] \]

are necessary to make a pasta dish. In this case, starting from state 0 in the trajectory, the algorithm checks for the state in which an action from the action list is executable from. If such state is found, the search process continues from the next state in the trajectory and this time looks for the next action in the action list. Once all actions are found, the algorithm returns ‘success’ along with the initial and final states, otherwise ‘failure’. Figure 3.4 shows an activity recognized with a sequence of actions.

![Figure 3.4: Activity recognized by a sequence of actions \([a_0, a_1, a_2]\).](image)

**Complexity of ActRec Algorithm**

The ActRec algorithm consists of two main parts; An ASP-algorithm that is external to system and a inference part that is used to recognize the activity (Figure 1.3. In order to infer about an activity, the system interacts with the ASP-algorithm and then feeds the obtained answer sets to the recognizer. Therefore, to calculate the complexity of whole system, we should take into account the complexity of ASP-algorithm and the recognizer. Deciding whether a given program has an answer set is NP-Complete [28]. For the recognizer part, we have a trajectory of length \(m\) and for each state \(s\) in the trajectory we check that if a set of fluents \(F\) or a set of actions \(A\) is subset of the state \(s\). In other words, the most expensive operation is the subset operation, i.e.

\[ O(m \times \text{subset operation}) \]

The subset operation, for two sets including \(n\) items could be completed under \(O(n^2)\) in the worst case if two sets are not ordered. Therefore, the recognition part is of the order of \(O(m \times n^2)\). The complexity of the ActRec algorithm is then NP-Complete.

**3.2.1 Activity Selection**

The ActRec algorithm described in the previous section returns a value for each trajectory. The returned value is one of values: recognized, uncompleted or notrecognized. Usually, a ASP program has several, say \(n\) answer sets. Each answer set in our approach represents a trajectory. Therefore we will obtain \(n\) results from the algorithm rather than one to infer an activity from our program. Hence we need to treat the results in a way that allows us to draw conclusions about the activity. We propose a probability method \(P(A)\) to aggregate the returned values. In this method, we sum up the total times that the algorithm returns recognized for trajectories and then divide it by the total number of trajectories. The
Algorithm 1 ActRec algorithm for activity recognition

function ActRec(Trajectory, initConditions, goalConditions, actionList)
    allTrajectories = getAllASP()
    for trajectory in allTrajectories do
        ActivityByFluents(Trajectory, initConditions, goalConditions)
        ActivityByAction(trajectory, actionList)
    end for
end function

function ActivityByFluents(Trajectory, initConditions, goalConditions)
    startState = null;
    finalState = null;
    for i=0 to size(Trajectory) do
        if startState == null then
            if initConditions ⊆ S_i then
                startState = S_i;
                continue;
            end if
        end if
        if startState != null and finalState == null then
            if goalConditions ⊆ S_i then
                finalState = S_i;
                break;
            end if
        end if
    end for
    if startState == null then
        return NOTRECOGNIZED;
    else
        if finalState == null then
            return UNCOMPLETE
        else
            return RECOGNIZED
        end if
    end if
end function

function ActivityByAction(trajectory, actionList)
    startState = null
    prevState = null
    for action in actionList do
        for i=0 to size(Trajectory) do
            actionSet = getActions(S_i)  # get executable actions in state S_i
            if startState == null then
                if action ⊆ actionSet then
                    startState = S_i; prevState = S_i
                    break
                end if
            end if
            if action ⊆ actionSet then
                startState = S_i; prevState = S_i
                break
            end if
            if action ⊆ actionSet then
                prevState = S_i
                break
            else
                return NOTRECOGNIZED
            end if
        end for
    end for
    return RECOGNIZED
end function
equation is as follows.

\[ P(A) = \frac{\sum_{i=1}^{n} V(T_i, A)}{n} \]  

(3.1)

Where \( A \) is the activity, \( n \) is total number of answer sets or trajectories, and \( T \) represents the trajectory. The function \( V(T, A) \) returns 1 if activity \( A \) is recognized within trajectory \( T \) and 0 otherwise. In chapter 5 we will elaborate more on activity recognition based on this simple equation.
Chapter 4

Description of ActRec System

In this chapter, we introduce the ActRec system. Basically, the ActRec system consists of three main parts: 1) a mapper, 2) a solver and 3) an activity recognizer. The work flow is as follows. First, the mapper converts the user input $C_{TAID}$ program into a logic program solvable by a solver. Then, the solver interacts with an ASP-solver and passes a logic program to the solver. The output of the solver then is passed to the recognizer for activity recognition. Figure 4.1 shows the work flow of the program. The following sections discuss the system requirements as well as a detailed description of three parts is presented.

![Figure 4.1: Work flow of ActRec System.](image)
4.1 System requirements

The ActRec system is a cross-platform solution and is developed in Python language. It could be run on different platforms as long as a suitable ASP-solver capable on running on the desired platform is provided. We used Clingo [25] as ASP-solver which could be found in several builds for different platforms. Additionally, SQLite RDBMS is used for managing the data in database. System requirements could be listed as follows:

- Operating system
- Python v. 2.7
- SQLite RDBMS
  - sqlalchemy library for Python
- Clingo ASP-solver
- Graphviz for output visualization (optional)
  - PyDot library for Python.

4.2 System Parts

The ActRec system consists of three parts: the mapper which converts the action specifications and observations in $C_{TAZD}$ into an ASP program, a ASP-solver to solve the converted program and recognizer to infer about the activities. Each part is separately described in this section with detail.

4.2.1 Mapper

The mapper module itself consists of three component: a database to store and retrieve rules and observations, controller which provides methods and functions to deal with the database, and a mapper which manages the mapping of rules and observations stored in the database into an ASP program. Figure 4.2 shows three parts of the mapper module.

Figure 4.2: Different components of the Mapper module.
4.2. System Parts

Database

The database is the central data storage for the rules and observations which are used for capturing the environment specifications. It includes tables for storing fluents, actions, various rules and constraints defined in $C_{TAID}$ language as well as world observations of the form defined in Section 2.2.1. Figure 4.3 depicts the logical view of the ActRec system’s database.

controller

Controller provides methods and tools for dealing with the database. It includes function to add, remove and update database tables. In other words, it serves as an interface between data layer (database) and the mapper.

Mapper

Mapper is responsible for converting the rules and observations stored in the database which are in $C_{TAID}$ language into an ASP program. Using methods included in the controller
interface, it interacts with the database and translates a $CT_{ATD}$ specification into an ASP program. For example, a dynamic rule of the form defined in 2.2.1 like:

\[
\text{vessel\_water\_boil} \ <\ 	ext{causes} > \ 	ext{vessel\_has\_boiled\_water} \\
\quad <\text{if} > \ 	ext{vessel\_water\_full}, \text{heat\_on}, \text{vessel\_on\_heat}
\]

is translated to its equivalent in ASP as:

\[
\text{holds(vessel\_has\_boiled\_water}, \ T+1) : \text{- holds(occurs(vessel\_water\_boil), } T), \\
\quad \text{holds(vessel\_water\_full}, T), \\
\quad \text{holds(heat\_on}, T), \\
\quad \text{holds(vessel\_on\_heat}, T), \\
\quad \text{fluent(vessel\_water\_full),} \\
\quad \text{fluent(heat\_on),} \\
\quad \text{fluent(vessel\_on\_heat),} \\
\quad \text{fluent(vessel\_has\_boiled\_water),} \\
\quad \text{action(vessel\_water\_boil), time(T), time(T+1).}
\]

Once the ASP program is generated, it is ready to be solved by the solver.

### 4.2.2 Solver

Solver module is responsible for solving ASP programs and parsing the result-set which will be used for activity recognition. The solver module accepts the generated ASP program from the Mapper and interacts with the asp-solver program to obtain the answer sets of the ASP program. If such result set(s) exist, then solver module will parse the output and make it ready for the recognizer module.

### 4.2.3 Recognizer

Once the result-set of the ASP-solver is parsed, they will be passed to the recognizer for activity recognition. Result-set should be prepared in the format of a trajectory described in Section 2.2.1. Recognizer module includes several functions that handle the task of representing answer sets in the trajectory format. The ActRec algorithm is then invoked to perform the task of activity recognition.

### 4.3 Data Flow

#### 4.3.1 Data Input

Activity recognition process includes several data input processes. First, all fluents and actions required to describe the environment should be defined. An example of such fluents and actions for our toy example of making pasta could be as follows.

**Fluents:**

- 'heat\_on',
- 'vessel\_on\_heat',
- 'vessel\_water\_full',
4.3. Data Flow

Fluents:

'vessel_pasta_full',
'pasta_cooked',
'vessel_sauce_full',
'pasta_sauce_mixed',
'vessel_has_boiled_water',
'pasta_water_drained',
'pasta_ready'

Actions:

'vessel_water_boil',
'pasta_boil',
'pasta_cooked_drain',
'pasta_sauce_add'

This process is performed via a user interface. See Figure 4.4.

![Figure 4.4: Adding fluents to the system.](image)

After adding fluents and actions to the system, we can now add desired rules and constraints in order to describe the environment with action language. Rules and constrains can be defined through a dialog interface as shown in Figure 4.5.

Now that the environment is formulated by rules and constraints, we can specify which activities we are interested in to be recognized. Desired activities could be introduced to the system via another interface. A screen-shot of the adding activity to the system is shown in Figure 4.6.

Now we can run the system and check whether an activity is performed within the formulated settings of the environment. If the activity is recognized, a trajectory showing different states of the trajectory with start and final states along with some detailed information on the activity will be shown. Additionally, the output trajectory could be represented as a graph diagram if the additional (optional) system requirements, e.g GraphViz program, are present. A visual description of the data flow is represented in Figure 4.7. Figure 4.8 depicts terminal output for a recognized activity in two steps.
Figure 4.5: Defining rules and constraints to the system.

Figure 4.6: Adding a desired activity to the system to check for recognition.

4.4 The Code

The source code of the program is available on the project github account at https://github.com/spartonia/ActRec.

Figure 4.7: Data flow of the ActRec system.
4.4. The Code

Figure 4.8: Terminal output for a recognized activity with two states.
Chapter 5

Evaluation

This chapter presents an empirical evaluation of the ActRec system and an informal discussion about this evaluation. Additionally, we outline how to interpret the data and infer complex activities from the obtained results using data mining techniques. Future work and suggestions for extending the current work are also discussed in this chapter.

5.1 Experiments

We performed experiments with some scenarios on the proposed ActRec system. Basically, we experimented two scenarios; ‘making pasta’ and ‘making coffee’. In the next section, we attempt to specify in a declarative way the environment with fluents and actions. The next section then represents the results obtained from the ASP solver for the scenarios.

5.1.1 Formulating the environment

Table 5.1 shows fluents and actions used in the experimented scenarios. Needless to say, these fluents and actions are up to the users and one may add or remove any arbitrary number of those to formulate the environment for both scenarios. We can now show how these two scenarios can be represented as a domain description in $\mathcal{CT}_{ATDP}$. For sake of simplicity, we represent the steps in details for the making coffee scenario. Similar approach could be applied to the scenario of making pasta.

Our knowledge about making coffee gives rise to the following dynamic causal rules.

\[
\text{coffe}_m\text{.activate} < \text{causes} > \text{coffe}_m\text{.is.on} < \text{if} > \neg(\text{coffe}_m\text{.is.on}).
\]

\[
\text{coffe}_m\text{.add_water} < \text{causes} > \text{coffe}_m\text{.has_water} < \text{if} > \neg(\text{coffe}_m\text{.has_water}).
\]

\[
\text{coffe}_m\text{.add_coffe_powder} < \text{causes} > \text{coffe}_m\text{.has_powder} < \text{if} > \neg(\text{coffe}_m\text{.has_powder}).
\]

Additionally, a static causal rule could be added as follows.

\[
\text{coffe}_m\text{.has_old_coffe_powder} < \text{if} > \text{coffe}_m\text{.has_water}, \text{coffe}_m\text{.has_powder}, \text{coffe}_m\text{.is.on},
\neg(\text{coffe}_m\text{.has_old_coffe_powder}), < \text{if} > \text{coffe}_m\text{.has_powder}.
\]
Table 5.1: Fluents and actions used in each scenario.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Fluents</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Making Coffee</td>
<td>coffe_m_is_on, coffe_m_has_powder, coffe_m_has_old_coffe_powder, coffe_m_has_water, coffe_m_is_jug_removed</td>
<td>coffe_m_activate, coffe_m_add_coffe_powder, coffe_m_remove_coffe_powder, coffe_m_add_water, coffe_m_add_jug, coffe_m_remove_jug</td>
</tr>
<tr>
<td>Making Pasta</td>
<td>heat_on, vessel_on_heat, vessel_water_full, vessel_pasta_full, pasta_cooked, vessel_sauce_full, pasta_sauce_mixed, vessel_has_boiled_water, pasta_water_drained, pasta_ready</td>
<td>vessel_water_boil, pasta_boil, pasta_cooked_drain, pasta_sauce_add</td>
</tr>
</tbody>
</table>

\[ neg(coffe_m_has_powder) < if > coffe_m_has_old_coffe_powder. \]

The latter two rules ensure that \( coffe_m_has_powder \) and \( coffe_m_has_old_coffe_powder \) cannot be true at the same time.

Now we can define a set of observations for our scenario. The initial state can be defined by the following fluent observations as follows.

\[ neg(coffe_m_is_on) < at > 0. \]

\[ neg(coffe_m_has_water) < at > 0. \]

\[ neg(coffe_m_has_powder) < at > 0. \]

For a time bound of \( t_{\text{max}} = 5 \), we obtain 4 possible answer sets. Again, it worth mentioning that by adding and/or removing some fluents, actions and rules to the scenario we may obtain different number of answer sets. The next section discussed the results of applying ActRec algorithm to the obtained answer sets.

### Applying ActRec Algorithm

Now that we obtained answer sets from the ASP solver, we can define and infer activities with our algorithm. As mentioned earlier in Chapter 3, answer sets are mapped into trajectories before feeding into the ActRec algorithm for recognition. In order to infer about an activity, we first need to define what we expect from an activity. In other words, initial and final states of the activity needs to be determined. For the scenario of making coffee, we defined the initial state as follows:
5.2 Discussion and Interpretation of Data

\[
\text{initialState} = [\text{coffee}_m\text{.has}_\text{powder}, \text{coffee}_m\text{.has}_\text{water}]
\]

And final state as

\[
\text{finalState} = [\neg(\text{coffee}_m\text{.is}_\text{jug}\text{.removed}), \text{coffee}_m\text{.has}_\text{old}_\text{coffee}_\text{powder}]
\]

<table>
<thead>
<tr>
<th>Scenario</th>
<th>#Answer Sets</th>
<th># Recognized</th>
<th>#Ongoing</th>
<th>Failed</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Making Pasta</td>
<td>618</td>
<td>16</td>
<td>24</td>
<td>578</td>
<td>0.025</td>
</tr>
<tr>
<td>Making Coffee</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 5.2: Results of running ActRec algorithm on two scenarios.

Since we defined the activity, we can run our ActRec algorithm to see whether we can recognize the defined activity in any on the obtained answer sets. For the scenario of making coffee, we can see that one activity is recognized given initial and final states. In addition, three trajectories include ongoing activities. Table 5.2 shows the results of running the algorithm for both scenarios in more details. In the next section we discuss how to analyze the results in details.

5.2 Discussion and Interpretation of Data

In the previous section, we illustrated how we can run the ActRec system to recognize an activity. The system outputs a value showing the probability of the recognized activity. Although the obtained probability is valuable, but it seems like a mere number and nothing more. In order to be benefited from the system, we describe some approaches for interpreting the results. One way is to aggregate the results of activity recognition for several activities. For instance, we can run the system to recognize activities such as: ‘cooking’, ‘walking into kitchen’, ‘doing dishes’, etc. within a limited time period, say 30 minutes, for example. We are then able to draw some conclusions based on the collected result set. Figure 5.1 depicts the idea of recognizing several activities over a limited time duration.

As we can see from the Figure 5.1, we can see that certain activities are comprising groups of activities as they are close together. We can use data mining techniques like clustering to detect groups. Clustering is a form of unsupervised learning that takes unlabeled data and places objects of the data in groups in such a way that objects of the same group are more similar in some senses to each other than to those in other groups. Generally, clustering is used when the data we are dealing with has no label. In other words, alike our data, we do not already know which object of the data belongs to what group. One interpretation for such groups are that activities inside a certain group resemble a complex activity. For instance, if activities such as ‘walking into kitchen’, ‘making coffee’, ‘opening fridge’ and ‘sitting’ fall into one group, we may infer that a complex activity resembling ‘eating breakfast’ is occurred.

We can further extend the idea of grouping activities for longer duration. If we run the system to recognize activities, say for a week or month, we can even figure out more interesting results. Suppose than we record the results of running the system for detecting certain activities for one week (see Figure 5.2). Because of the time limit and also the scope of the thesis, here we used synthetic data for illustrating the concept. Real data from running the ActRec system could be used in the similar fashion. We use clustering to detect
group of activities falling into similar categories. As we can see from figure 5.2, a person in this case repeats certain patterns during the time period. By this means, we can infer some behaviors of the person performing the activities. For example, if the group of activities is recognized as taking drugs, we may infer that the person suffers from some illness. Then, if we detect an anomaly in the recorded pattern, for instance we don’t detect the activity of taking drug in a certain day, we may infer that the person forgets taking drugs that day.
5.3 Future Work

For the future work, we propose extending the use of data mining techniques for activity recognition as a wrapping layer to the proposed ActRec system. This will enable us to infer about the behaviors, which are collection of activities, rather than activities. Besides using unsupervised learning, supervised learning methods also could be used for several purposes. One possibility is to build predictor models in order to predict the behavior of users. By this means we are able to predict the emerging activities by taking into account the history of the performed activities by user.

Another interesting point for the future work is to use benchmarks for the activity recognition. These benchmarks are mostly useful with data mining techniques for activity recognition. However, fitting those data into our system will enable us for drawing stronger conclusions considering both data driven and our approach. Some of those benchmarks could be found at [29].
Chapter 6

Conclusion

Different techniques have successfully been applied to infer human activity, including machine learning and data mining [5]. These data driven techniques work well within a particular domain and situations in which they are initially set in. However there are some drawbacks with these techniques such as being domain dependent and need for large amount of data collection and annotation.

Against this background, we attempt to build a complex activity recognition technique that not only rely on the sensory data but also involve domain knowledge as well. We proposed and developed ActRec system, a novel activity recognition algorithm to tackle the problem of recognizing human activities. We approached the problem of recognizing human activity through non-monotonic reasoning and logic programming and used action languages, namely $\mathcal{C}_{TAID}$ for modeling the environment. $\mathcal{C}_{TAID}$ provides several features for describing the environment in a declarative action specification. The problem instance is described in action language $\mathcal{C}_{TAID}$ and then converted into ASP program by mapping. We have introduced a novel definition of activity by considering the answer sets of an ASP program. The answer sets which are obtained from a ASP program are mapped into trajectories in our proposed approach. Our implemented ActRec system takes the answer sets of the ASP program in the form of trajectories and tells us whether an activity is performed or not. We attempted to answer the following questions in our work.

- Which activity was performed?
- How was the activity performed?
- How can we be sure that the activity was performed?

The ActRec system attempts to answer the above questions. By introducing the specification of the desired activity to the system, we can run the recognition task to see whether a certain activity is performed or not. The trajectory of the activity gives us an explanation of the activity, in case the activity is recognized in a trajectory. A probability value, after performing the activity recognition task, determines the confidence level of occurrence of a certain activity.

Additionally, we outline the idea of inferring about the behaviors with the use of data mining and machine learning techniques. This ideas could be extended in the future work.
Chapter 7

Acknowledgments

I would like to thank Dr. Juan Carlos Nieves Sánchez for his valuable guidance, patience and sport throughout this research. I have been extremely lucky to have a supervisor who cared so much about my work.

My sincere appreciation goes to prof. Frank Drewes and Yvonne Lòwstedt, for their patience and support during my studies at Umeå University.

Last but not least, my parents and family receive my deepest gratitude and love for their dedication and the many years of support during my life that provided the foundation for this work.
References


Appendix A

Source Code

The source code for the main algorithm is presented in this Chapter. The source code of the whole system is available on Github repository at http://github.com/spartonia/actrec.

def actRec(initState=[], finalState=[], actionList=[], printTrajectory=True, printAllStates=True, printAllActions=True):
    # if not initState and not finalState:
    cnt_recognized = 0
    cnt_uncomplete = 0
    cnt_notrecognized = 0
    cnt_recognized_by_action_list = 0
    cnt_notrecognized_by_action_list = 0
    if actionList:
        for i, j in actionList.getItems():
            print i, j

    for i, answerSet in get Answers():
        print "Recognizing activity for answer:", i
        status = activity_recog_by_fluents(answerSet, initState, finalSet,
            printTrajectory, printAllStates)
        if status == RECOGNIZED:
            cnt_recognized += 1
        elif status == NOTRECOGNIZED:
            cnt_notrecognized += 1
        else:
            cnt_uncomplete += 1
        status = activity_recog_by_actions(answerSet, actionList,
            printTrajectory, printAllActions)
        if status == RECOGNIZED:
            cnt_recognized_by_action_list += 1
        else:
            cnt_notrecognized_by_action_list += 1

    print "Summary:"
    print "By fluents:"
    print "\tTotal recognized activities: ", cnt_recognized
    print "\tTotal uncomplete activities: ", cnt_uncomplete
    print "\tTotal not recognized trajectories: ", cnt_notrecognized
    print "\tTotal activity performed with the probability of: ", (cnt_recognized / (cnt_recognized+cnt_notrecognized+cnt_uncomplete))
    print "By action set:"
    print "\tTotal recognized activities: ", cnt_recognized_by_action_list
    print "\tTotal uncomplete activities: ", cnt_notrecognized_by_action_list

43
print "Activity performed with the probability of: ", (cnt_recognized_by_action_list/(cnt_recognized_by_action_list+cnt_notrecognized_by_action_list))

def activity_recog_by_actions(answerSet, actionList, printTrajectory=False, printAllActions=False):
    """Recognize activity by checking whether a sequence of actions performed."
    
    states = {}
    for state, actionSet in make_action_world(answerSet):
        states[state] = actionSet
    states = collections.OrderedDict(sorted(states.items()))
    actionList = collections.OrderedDict(sorted(actionList.items()))
    if (not actionList) or (not states):
        return NOTRECOGNIZED # NO action list provided
    if printAllActions:
        print "All actions (World): ",
        for state, actionSet in states.items():
            print "State: " + state + ": ", actionSet
        print
    startState = None
    prevState = None
    for action_no, action in actionList.items():
        # print action_no, action
        for state, actionSet in states.items():
            # print startState, 'startState'
            if startState == None:
                if action.issubset(actionSet):
                    startState = state
                    prevState = state
                    if printTrajectory:
                        print "action", action_no, action
                        print "is subset of ", state, actionSet
                    # print 'Start State:', startState
                    break
            else:
                continue
            if startState != None and state <= prevState:
                continue

            if action.issubset(actionSet):
                prevState = state
                if printTrajectory:
                    print "action", action_no, action
                    print "is subset of ", state, actionSet
                break
            else:
                if printTrajectory:
                    print "action", action_no, action
                    print "is not a subset of ", state, actionSet
                    print \NOTRECOGNIZED\ # activity not found

            # print action_no, state
            # print len(actionList), prevState, startState
            if printTrajectory:
                print \RECOGNIZED\" print "Activity started at state '{0}' and finished at state '{1}'.\n format(startState, startState) # +len(actionList)-1
def activity_recog_by_fluents(answerSet, initSet, finalSet, printTrajectory=False, printAllStates=False):
    """ Recognize activity by checking whether Initial and final state conditions are met."""
    # what if initSet and final sets are empty?
    states = {}
    for state, fluentSet in make_fluent_world(answerSet):
        states[state] = fluentSet
    # sort states
    states = collections.OrderedDict(sorted(states.items()))
    if printAllStates:
        print "All States: 
        for state, fluentSet in states.iteritems():
            print "State ", state, ': ', fluentSet
        print "Initial State: ", [i for i in initSet]
        print "Goal State: ", [i for i in finalSet]
        print

    initState = None
    finalState = None
    for state, fluentSet in states.iteritems():
        if initState == None:
            if initSet.issubset(fluentSet):
                initState = state
                continue
        if initState != None and finalState == None:
            if finalSet.issubset(fluentSet):
                finalState = state
                break
        if initState != None and finalState != None and printTrajectory:
            for state, fluentSet in states.iteritems():
                if state == initState:
                    print "State:", state, '---'>, [i for i in fluentSet]
                    print
                    continue
                print "\t\t\t\t\t\t---" 
                print "\t\t\t\t\t\t| |----" 
                print "\t\t\t\t\t\t\\|||" 
                print "\t\t\t\t\t\t\\//" 
                print "\t\t\t\t\t\t\n/" 
                print "State:", state, '---->', [i for i in fluentSet]
                print
                if state == finalState: break
        if initState == None:
            print "Activity not recognized! Initial state conditions not met."
            return NOTRECOGNIZED
        else:
            print "Activity started at state {} (Initial State).".format(initState)
        if finalState == None:
            print "But not finished yet! (ongoing, final state conditions not met)."
            return UNCOMPLETE
        else:
print "and terminated at state {0}".format(finalState)
return RECOGNIZED