Integrating Logic Programming with Description Logic Reasoning and Sensor Observation Management for Mobile Devices

by

Dustin Troy Cline

(Under the Direction of Frederick W. Maier)

Abstract

The ubiquity of mobile devices has opened new research opportunities for knowledge-based systems. The purpose of this work is to integrate logic programming with description logic reasoning and sensor observation management for mobile devices. Two Prolog libraries were developed, each facilitating access to a different form of external information. One library leverages the OWL API and specialized reasoners to compute inferences from ontologies expressed in the Web Ontology Language while the other automatically acquires, stores, and manages sensor observations reported by on-board sensors of Android-powered devices. The libraries are developed independently of one another and are designed to function in isolation. However, they are also compatible. A physical activity recognition task demonstrates their collaborative use.

Index words: Logic programming, Prolog, description logics, Web Ontology Language, sensors, mobile devices

Integrating Logic Programming with Description Logic Reasoning and Sensor Observation Management for Mobile Devices

by

Dustin Troy Cline

B.S., Valdosta State University, 2012

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2015

Integrating Logic Programming with Description Logic Reasoning and Sensor Observation Management for Mobile Devices

by

Dustin Troy Cline

Approved:

Major Professor:	Frederick W. Maier
Committee:	Walter D. Potter William Hollingsworth

Electronic Version Approved:

Julie Coffield
Interim Dean of the Graduate School
The University of Georgia
May 2015

Acknowledgments

I would like to extend many thanks to my major professer Dr. Maier who helped me greatly over the course of completing this thesis. I would like to thank Drs. Potter and Hollingsworth for serving as members of my committee.

I would also like to thank Sai Balakavi with whom I shared an office space. It was through discussions with him and his discovery and sharing of a particular dataset that I was able to wrap everything up nicely.

I would like to thank Kelly Storm for insisting that I finish this degree before pursuing a career. Who knows if I would have finished in a timely manner (if at all) had I done otherwise.

And of course, I would like to thank my family – my parents for their financial and emotional support throughout my academic career as well as my grandmother for her support and delicious baked goods.

Lastly, I would like to thank Amelia Rhodes. Without your suggestion that I consider pursuing higher education, I would have never started this degree. Without your loving support and encouragement, I might not have ever finished it. Thank you for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Mobile devices are becoming more prevalent in the daily lives of people. Their increasing popularity can be attributed to a combination of portability and versatility. They are also becoming more powerful allowing them to replace desk-bound workstations and cumbersome laptops for many tasks. Additionally, most mobile devices are equipped with a variety of sensors such as accelerometers, GPS, and thermometers that report observations regarding the current state of the device and its environment. Currently, Android is the most popular platform for mobile devices based on reports of world-wide sales [1]. Due to their popularity, portability, and versatility, mobile devices open a new venue to explore knowledge-based systems.

The purpose of this work is to develop a framework integraging logic programming (LP) with description logic (DL) reasoning and sensor observation management for mobile devices. Two Prolog libraries have been developed for an Android-compatible Prolog environment, one for each task. One library leverages preexisting tools and specialized reasoners to compute inferences from ontologies expressed in the Web Ontology Language (OWL) [2], a now

standard DL-based formalism for expressing ontologies. The other library automatically aquires and manages sensor observations reported by on-board sensors of Android-powered devices. Both libraries are developed independently of one another and are designed to function in isolation. However, they are designed to be compatible and can be used together with ease.

1.2 Integrating Prolog with OWL for Android devices

Two of the most prominant formalisms for knowledge representation and reasoning (KRR) are rule-based systems, exemplified by logic programming (LP) [3], and description logics (DLs) [4]. Logic programming has been thoroughly researched since the 1970s. Though DLs are newer to appear, they have also been extensively researched for the past few decades, beginning in the early 1990s. The two formalisms have different characteristics, in some cases making one more suitable for a given application than the other. They have different syntax and semantics, making integration difficult. Nevertheless, integration would be beneficial, and there is a large body of work dedicated to the pursuit [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Much of this interest is due to the plan to integrate the two formalisms to enhance the KRR capabilities of the Semantic Web [16]. It was proven early on that a naive combination leads to undecidability [6]. While there are syntactic differences, the more fundamental problem is that the two formalisms possess very different semantics, as discussed in [12].

Among the most significant differences between the two formalisms is that LP is non-monotonic while DLs are monotonic. The addition of a new fact into the knowledge base of a logic program may decrease the number of derivable consequences. On the other hand, the monotonicity of DLs means that the addition of a new axiom into a DL ontology cannot decrease the number of derivable consequences (though it may cause the ontology to become inconsistent).

Additionally, both formalisms make opposing assumptions regarding unknown knowledge. LP makes the closed-world assumption (CWA), which means that a statement is assumed to be false by default if it cannot be proven true. Conversely, DLs make the open-world assumption (OWA). If a statement cannot be proven true it is not considered to be false (or true). Similarly, if it cannot be proven false it is not considered to be true (or false).

Furthermore, LP makes the unique name assumption (UNA), meaning that syntactically different terms are interpreted to denote different entities in the domain. DLs, however, do not make the unique name assumption. The same entity can be referenced using different identifiers and axioms regarding the equality or inequality of entities must be explicitly asserted into the ontology.

OwlLib is an Android-compatible Prolog library developed during the course of this thesis that integrates logic programming with description logic reasoning regarding OWL ontologies. It maintains the OWL ontologies separately from the Prolog knowledge base and leverages external reasoners to compute inferences regarding the knowledge expressed in the ontologies while the Prolog inference engine is used to perform reasoning over the Prolog knowledge base. The inferences computed by the reasoners are made available to the Prolog program by the OwlLib predicates that query the reasoners. OwlLib also provides predicates and terms for representing OWL constructs and manipulating ontologies by adding and removing axioms from them.

1.3 Integrating Prolog with Android sensors

Most Android devices are equipped with an array of sensors that report observations regarding the current physical state of the device and its environment. The observations are rich with information regarding what the user is doing at a given time. The data associated with the observations provide *context* [17] which can be used to create *context-aware systems* [18],

the latter constituting a relatively recent research field.

SensorLib is an Android-compatible Prolog library that has been developed during the course of this thesis to aquire and manage sensor observations and make them available to Prolog programs. SensorLib mitigates the low-level requirements of obtaining sensor observations through the implementation of an automated mechanism for acquiring, storing, and managing sensor observations in the form of *observation managers* (OMs). OMs implement the sliding window technique popular with data stream management systems [19] in which only a limited amount of the most recently received observations are maintained. The functionality provided by SensorLib facilitates the implementation of context-aware systems.

Android does not permit for sensors to be directly polled for their observations. Instead, the process of aquiring sensor observations involves defining "listener" objects that must be registered with particular sensors in order to receive observations when they are reported. Also, sensor observations are reported one at a time and there is no built-in funtionality for handling them once they arrive. Furthermore sensors can report observations rapidly, producing a steady stream of data. These factors make the utilization of sensor observations in Prolog challenging. SensorLib is designed to overcome them.

1.4 Contributions

The contributions of this thesis are as follows.

OwlLib, an Android-compatible Prolog library for working with OWL ontologies and reasoners, is developed and presented. OwlLib provides a number of Prolog predicates and terms for performing a range of tasks such as managing OWL ontologies and reasoners, querying OWL reasoners, and manipulating OWL ontologies. OwlLib leverages Java-based Android-compatible tools to implement these predicates and terms, specifically the OWL API [20] for working with ontologies and reasoners as well as the popular JFact

[21] and ELK reasoners [22] for computing inferences regarding ontologies.

OwlLib allows for OWL reasoners to be queried from within a Prolog program in a variety of ways. OwlLib also allows for the consequences of a Prolog program to be added to an OWL ontology to make them accessible by OWL reasoners. In other words, knowledge flow is bi-directional rather than unidirectional. This functionality is inspired by related work that integrates logic programming with DLs called *dl-programs* [11, 14, 15].

OwlLib ensures that the opposing semantics of both formalisms are maintained by keeping the knowledge expressed in an ontology separate from that in the Prolog knowledge base as well as leveraging specialized reasoners to compute inferences regarding the ontology.

SensorLib, an Android-compatible Prolog library for working with Android sensors, is developed and presented. SensorLib abstracts away from the low-level requirements of obtaining sensor observations from Android sensors. It supplies observation managers (OMs) for the automated acquisition, storage, and management of reported sensor observations using the proven sliding window technique common in data stream management systems [19]. SensorLib provides access to the maintained observations from Prolog. Predicates are supplied for obtaining a copy (or clone) of a window of an OM so that the observations it maintains can be inspected. SensorLib also implements operations that compute statistical summaries (min, max, mean, etc.) of the observation values within a cloned window.

1.5 Outline of later chapters

The remainder of this thesis is organized as follows. Chapter 2 provides background information and related works. Details regarding the development of applications for Android are presented followed by descriptions of the types of sensors that can be found on an Android device. Logic programming, Prolog, and JPR (an Android-compatible Prolog environment

developed at the Institute for Artificial Intelligence at the University of Georgia) are discussed in detail, followed by DLs and OWL. Works related to OwlLib and SensorLib are also presented.

OwlLib is outlined and presented in chapter 3. The structure of the library is discussed, illustrating how it relates to JPR, the OWL API, and external reasoners. The ontology and reasoner management functionality of OwlLib are discussed followed by an explanation of the manner in which OWL entities and constructs are represented in OwlLib. The various ways of querying OWL reasoners are discussed in detail, including implementation details regarding of the bi-directional knowledge flow between a Prolog program and an OWL ontology. Lastly, issues regarding the integration of Prolog and OWL are discussed.

SensorLib is outlined in chapter 4. Details regarding the aquisition of sensor observations using the Android Software Development Kit (SDK) [23] are presented and the structure of SensorLib is discussed. The observation managers provided to aquire, store, and manage sensor observations are discussed in detail. The methods for accessing the observations from a Prolog program are discussed as well as the manner in which observations are represented in SensorLib. The built-in operations for computing statistical summaries of windows of observations are presented.

Chapter 5 demonstrates how both libraries can be used collaboratively along with Prolog rules to perform tasks related to physical activity recognition on an Android device. The data set of sensor observations of physical activities and the training instances derived from it are discussed. The creation of a decision tree capable of performing physical activity detection is outlined along with the technique used to produce a comparable tree of much smaller size. The transformation of the tree into equivalent Prolog rules is discussed. The use of SensorLib to aquire sensor observations and compute statistical summaries of them is presented. The use of OwlLib to create a "semantic log" of detected activities is discussed.

Chapter 6 provides a summary and outlines possible directions for future work.

Chapter 2

Background and Related Work

2.1 Introduction

This chapter provides background information regarding Android, Android sensors, logic programming, Prolog, JPR, description logics, and OWL, necessary for the presentation of OwlLib and SensorLib later in the thesis. Works related to OwlLib and SensorLib are also presented. Details regarding the Android platform and Android application development are presented in section 2.2.1 and section 2.2.2 discusses Android sensors. Section 2.3.1 briefly describes logic programming and Prolog and section 2.3.2 presents some details regarding JPR, the Android-compatible Java-based Prolog interpreter for which OwlLib and SensorLib are developed. Description logics (DLs) and the Web Ontology Language (OWL) are outlined in section 2.4. Section 2.5 describes preexisting work related to the integration of logic programs with DL reasoning. Section 2.6 outlines work related to the use of sensor observations for activity recognition.

2.2 Android and Android sensors

2.2.1 Developing applications for the Android platform

The Android [24] operating system (OS) is a Linux-based system common on many mobile devices. Android applications are written using the Java [25] programming language. Each Android application runs as a seperate process [26] on the device to isolate it from other applications. The two main types of application components are Activities and Services. An Activity is a visible screen of an application with which the user interacts using widgets. A Service is intended for performing long running tasks in the background and therefore does not have a user interface. Activies are implemented by the `Activity` Java class and Services are implemented by the `Service` class, both of which extend the `Context` class in order to obtain global information regarding an application environment as well as gain access to system level services.

When an Android application is started, a single "main" thread of execution is started. The user interface (UI) runs on the main thread (sometimes referred to as the UI thread). All Java code executed from within an Activity or Service also runs on the main thread by default. Typically, computationally intensive tasks are performed on a separate thread to prevent hindering the responsiveness of the UI [26].

Android applications are written using the Java programming language. However, the Java libraries provided by Android Software Development Kit (SDK) [23] do not include a full implementation of the standard Java libraries provided by the Java Development Kit [25] which many third-party Java libraries leverage. This makes reusing existing third-party Java libraries difficult because any code that leverages any of the unsupported standard Java libraries are incompatible with Android.

2.2.2 Android sensors

The sensors on an Android device can be organized into categories based on the type of observations they report. Motion sensors monitor the movement of a device and measure acceleration and rotation forces. Position sensors monitor the orientation of a device and its proximity to other objects. Environmental sensors monitor the environment external to a device and can measure light exposure, temperature, and barometric pressure. Some of the sensors are implemented using hardware while others are implemented in software and derive their data using other sensors. A list of some of the various sensor types along with what they measure is provided in table 2.1

Table 2.1: Android sensors

Sensor (type)	Measures
Accelerometer (motion)	Acceleration force (including gravity) along the x, y, and z axes in m/s^2
Gyroscope (motion)	Rate of rotation around the x, y, and z axes in rad/s
Magnetic field (position)	Geomagnetic field strength along the x, y, and z axes in μT
Proximity(position)	Distance from object in cm
Ambient temperature (environmental)	Ambient air temperature in °C
Light (environemental)	Illuminance in lx

Different sensor types use coordinate systems with different frames of reference. Accelerometers, gravity sensors, gyroscopes, linear acceleration sensors, and magnetic field sensors use a device-oriented frame of reference as illustrated in figure 2.1. Other sensors use a world-oriented frame of reference. The X axis of the rotation vector sensor is tangential to the groud and points East, while the Y axis points towards magnetic north tangential to the ground, and the Z axis points upwards perpendicular to the ground. The geomagnetic rotation vector sensor uses the same coordinate system as the rotation vector sensor but will never rely on a gyroscope. The game rotation vector sensor does not rely on the geomagnetic

field at all and the Y axis does not point north but some other unspecified reference.

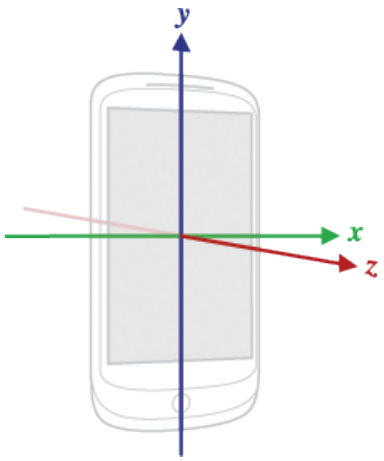

Figure 2.1: Android sensor coordinate system[1]

Android differentiates the sensors that detect motion, position, and the environment around a device from the "location providers" that detect the geographic location of a device. The geographical location can be provided by GPS or network location providers using cell towers and WiFi access points. While GPS location recognition is much more accurate, it is generally slower to aquire a location fix. Also, GPS does not work well indoors as it requires a view of the sky. Location providers are not part of the sensors framework though they are able to report changes in location in the same manner that the other types of sensors report their observations. It can be argued that since these location providers can detect the geographical location of a device, they can also be considered sensors.

The sensors on an Android device cannot be directly polled for their observations. Instead, each observation is delivered to listener objects through callback methods. A listener must be registered with a particular sensor in order to receive the observations reported by the sensor. A single listener can be registered for different sensors or multiple listeners can be registered for the same sensor. Upon listener registration, the developer is allowed to request the rate at which sensors report their observations, though this request may be disregarded

[1] `http://developer.android.com/reference/android/hardware/SensorEvent.html`

by the sensor. The majority of sensor types report observations repeatedly while others only report when the values they measure change[2]. When a sensor reports a new observation, the data regarding the observation is passed to each listener registered with that sensor. Registered listeners can be unregistered from their corresponding sensors in which case they will no longer receive reported observations. A standard practice is to unregister listeners when they are no longer needed or when an application loses focus in order to conserve device resources [27].

2.3 Logic programming, Prolog, and JPR

2.3.1 Logic programming and Prolog

Logic programming (LP) is a rule-based formalism that is commonly used for knowledge representation in artificial intelligence [3]. Knowledge is expressed in a logic program using if-then rules and queries to the program are solved using logical deduction. The account given here is based on definitions from [28] and [3].

A *logic program* is a set of rules represented as clauses of the form:

$$\texttt{h :- b}_1\texttt{, ..., b}_m\texttt{, not b}_{m+1}\texttt{, ..., not b}_n\texttt{.}$$

where $1 \leq \text{m} \leq \text{n}$, **h** and each $\mathbf{b}_i$ are literals (atoms or their negations), and **not** is negation-as-failure (NAF; also called *default negation*). **h** is called the head (or consequent) and the conjunction of $\mathbf{b}_i$s is called the body (or antecedant). The rule above can be read declaratively as "If $\mathbf{b}_1$ and ... and $\mathbf{b}_m$ and not $\mathbf{b}_{m+1}$ and ... and not $\mathbf{b}_n$, then **h**". Variable are allowed to appear in atoms. Variables appearing in the head of a rule are univerally quantified while those appearing in the body but not the head are existentially quantified.

[2]The light, proximity, humidity, and ambient temperature sensors only report observations when the values they measure change.

A *definite* logic program consists of rules containing no forms of negation (**not** and negative literals are not allowed). Definite logic programs can be interpreted under the classical semantics in which the unique minimal Herbrand model M_P of a definite program P contains all the atomic formulas entailed by P [28]. Logic programs containing negation require additional consideration.

A *normal* logic program consists of rules that may contain **not** but do not contain negative literals. An *extended* logic program allows both **not** and negative literals. The answer set semantics (also called the stable model semantics) [29] and the well-founded semantics [30] are popular approaches to handling the negations found in normal and extended logic programs.

Prolog is the earliest programming language designed for logic programs. A Prolog program consists of an ordered set of rules and facts (rules with no bodies) that constitute a knowledge base (KB).

```
human(bob).               % a fact
organism(X) :- human(X). % a rule
```

A Prolog interpreter is an inference engine that answers queries regarding the knowledge stored in the KB. The basic inference procedure implemented by Prolog interpreters is a backward-chaining depth-first search called SLDNF-resolution. A query Q is an ordered conjunction of goals g_1, ..., g_n. To solve Q, the inference engine searches the KB in a top-down fashion while attempting to resolve each goal g_i by unifing it with the head of each rule. When unification succeeds, the current goal g_i is replaced with the body of the rule (if it exists) and then resolution continues. If multiple rules resolve with the current goal, then a backtrack point is recorded. If a goal is preceeded by **not**, then it succeeds only if it cannot be resolved with a rule. If the inference engine is unable to resolve a goal, it backtracks to the most recently recorded backtrack point and tries again. If every g_i is able to be resolved, then Q succeeds. Otherwise, Q fails.

Prolog is not logically pure and has many procedural aspects that prevent it from being a fully declarative language. The depth-first search performed by resolution can get stuck in loops, as illustrated by the following rule:

```
a :- b, c, a.
```

and the query `?- a`. Resolution will attempt to solve `a` by first solving `b`, then solving `c`, and then solving `a` again. Furthermore, negation-as-failure is not equivalent to classical negation, and the built-in cut operator (`!/0`) allows for recorded backtrack points to be discarded, thereby affecting the evaluation of queries.

Nevertheless, Prolog is a Turing-complete language. In many practical applications, procedural operations such as reading input and writing output and performing mathematical calculations (which Prolog provides) are very much needed. Therefore, the failure of Prolog to be completely declarative is arguably not a drawback but rather an advantage and necessary in some cases.

2.3.2 JPR – Java PRolog

JPR[3] is a Java-based Prolog environment currently under development at the Institute for Artificial Intelligence of the University of Georgia. The motivation for the development of JPR is to provide an Android-compatible Prolog inference engine that is capable of interacting with Java. Both OwlLib and SensorLib are designed for use with JPR.

JPR defines Java classes representing Prolog terms, clauses, knowledge bases, and inference engines. JPR can be used from within Java program by creating an instance of an inference engine implemented by the class **`InferenceEngine`**. A query can be placed to the inference engine by calling its **`solve(String, List<Variable>)`** method in which the query is represented by a Java **`String`** object and the variables that are bound while the

[3] `http://www.ai.uga.edu/jpr`

query is being solved are placed in the Java **List** of JPR **Variable**s. The **solve** method evaluates to true if the query succeeds and false otherwise. The following code snippet demonstrates how to instantiate a JPR inference engine and submit a query to the inference engine:

```
// create a new JPR inference engine
InferenceEngine jpr = new InferenceEngine();
// create an empty list of variables
List<Variable> vars = new ArrayList<Variable>();
// submit a query to the inference engine, which binds
//   variables during the evaluation
boolean result = jpr.solve("human(X). ", vars);
```

Predicates that perform some action or otherwise require special handling (e.g., **write/1**) are defined in JPR via their own Java classes. When the JPR inference engine evaluates an instance of a Java-based predicate, rather than performing resolution, code in the implementation of the Java class that defines the predicate is executed. A JPR library is a collection of Java classes defining custom predicates and object factories used to create instances of the predicates at runtime. A library can be loaded into JPR in the following way:

```
InferenceEngine jpr = new InferenceEngine();
MyLibrary mylib = new MyLibrary();
jpr.termFactory().loadLibrary(mylib);
```

JPR allows references to instances of Java objects to be maintained by special terms called *reference terms*. Reference terms implement the **ReferenceTerm** Java interface and are treated as regular Prolog terms by JPR. The utility of reference terms comes from the the ability of Java-based predicates to access the Java object maintained by a reference term when it is received as an argument. A Java-based predicate can also "wrap" a Java object in a reference term and bind the term to an argument for use as output. Thus reference terms allow Java objects to be passed as terms in a rule or query.

2.4 Description logics and OWL

2.4.1 Description logics

Description logics (DLs) are a family of logic-based knowledge representation languages that are commonly used to express domain knowledge in the form of ontologies. A DL ontology is a knowledge base consisting of a set of axioms defined using symbols representing concepts (essentially, unary predicates of first-order logic), roles (binary predicates), and individuals (constants). The axioms are divided into a TBox ("terminological box") consisting of axioms defining subsumption relationships between concepts and an ABox ("assertional box") consisting of axioms defining assertions regarding specific individuals. There are a number of different DLs that can be obtained by permitting or restricting certain features. Below, the syntax and semantics of the basic DL $\mathcal{ALC}$ [31] are defined, based on material presented in [4] and [32].

A concept description describes a set of individuals. Complex descriptions can be built inductively with atomic concepts and atomic roles using concept constructors. For instance, the atomic classes Human and Female can be used to describe individuals who are a human and female using Human $\sqcap$ Female while individuals who are a human and not female can be described using Human $\sqcap \neg$Female. Similarly, individuals who are not female and have a child who is also a human can be described using Human $\sqcap \neg$Female $\sqcap \exists$hasChild.Human. Concept descriptions can be combined and nested in this fashion to construct increasingly complex concept descriptions.

Definition. ([32]) *Let N_C be a set of* concept names *and N_R be a set of* role names. *The set of* concept $\mathcal{ALC}$ descriptions *is the smallest set $\boldsymbol{C}$ such that*

$$\boldsymbol{C} ::= \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall r.C \mid \exists r.C$$

where $A \in N_C$ and $r \in N_R$.

A TBox contains general concept inclusion (GCI) axioms such as Human $\sqsubseteq$ Organism. A

symmetrical pair of GCIs ($C \sqsubseteq D$ and $D \sqsubseteq C$) is indicative of concept equivalence. That is to say, they describe the same individuals. The equivalence relationship between concepts can be represented using the abbreviation $C \equiv D$. For instance, $\mathsf{Man} \equiv \mathsf{Human} \sqcap \neg\mathsf{Female}$ means that individuals described by the complex concept description can also be described by the concept Man.

Definition. ([32]) *A* general concept inclusion *(GCI) is of the form $C \sqsubseteq D$ where C, D are concepts. A finite set of GCIs is called a* TBox.

An ABox can contain two types of axioms. *Concept assertions* assert that an individual is an instance of a given concept. *Role assertions* assert that a pair of individuals are connected by a given role name.

Definition. ([32]) *An* assertional axiom *is of the form $x : C$ or $(x, y) : r$, where C is a concept, r is a role name, and x and y are individual names. A finite set of assertional axioms is called an* ABox.

DLs are a fragment of first-order logic (FOL). Therefore, their semantics are that of FOL and they are interpreted under a model-theoretic semantics given in terms of interpretations.

Definition. ([32]) *An* interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *consists of a non-empty set $\Delta^{\mathcal{I}}$, called the* domain *of $\mathcal{I}$, and a function $\cdot^{\mathcal{I}}$ that maps every concept to a subset of $\Delta^{\mathcal{I}}$, and every role name to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that, for all concepts C, D and all role names r,*

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}, \qquad \bot^{\mathcal{I}} = \emptyset,$$
$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}, \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}, \qquad \neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$$
$$(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y : \langle x, y\rangle \in r^{\mathcal{I}} \text{and } y \in C^{\mathcal{I}}\},$$
$$(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y : \langle x, y\rangle \in r^{\mathcal{I}} \text{implies } y \in C^{\mathcal{I}}\}$$

where $C^{\mathcal{I}}$ ($r^{\mathcal{I}}$) is the extension of the concept C (role name r) in the interpretation $\mathcal{I}$ and if $x \in C$ then x is an instance of C in $\mathcal{I}$.

Definition. ([32]) *An interpretation $\mathcal{I}$* satisfies *(is a model of) a general concept inclusion $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; $\mathcal{I}$ satisfies a TBox $\mathcal{T}$ if it satisfies every GCI in $\mathcal{T}$. An interpretation $\mathcal{I}$* satisfies *an assertional axiom $x : C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$, and $\mathcal{I}$ satisfies an assertional axiom $(x, y) : r$ if $\langle x^{\mathcal{I}}, y^{\mathcal{I}}\rangle \in r^{\mathcal{I}}$; $\mathcal{I}$ satisfies an ABox $\mathcal{A}$ if it satisfies every axiom in $\mathcal{A}$.*

There are a number of inference problems that are common with DLs. Consistency checking involves determining whether an ontology is consistent. Satisfiability checking involves determining if a concept is satisfiable, meaning that it is not contradictory. Subsumption checking involves determining if all the individuals belonging to one concept also belong to another concept. Concept equivalence (resp. disjointness) checking involves determining if two concepts contain only the same (resp. none of the same) individuals. Membership checking involves determining if an individual belongs to a given concept. Relationship checking involves determining if two individuals are connected by a given role.

2.4.2 The Web Ontology Language

The Web Ontology Language (OWL) is a suite of knowledge representation languages for developing ontologies that is standardized by the World Wide Web Consortium (W3C). The most recent version of OWL, referred to as OWL 2, was adopted in 2009. Any future use of "OWL" in this document will refer to OWL 2 unless otherwise specified. The original purpose of OWL was to comprise the ontology layer of the Semantic Web [16] so that web resources could be annotated with semantic markup allowing machines that store, retrieve, and process the data to "understand" the intended meaning of the information. OWL has been used to express many ontologies in a wide range of fields. For example, the Semantic Sensor Network (SSN) ontology [33] is an ontology for describing sensors and their observations and OWL-Time [34] is an ontology for describing time using instants and intervals based on Allen's interval algebra [35]. OWL is chosen as the DL specification of focus for this thesis due to its expressiveness and acceptance along with the availability of tool support.

In OWL, concepts are referred to as *classes* and concept descriptions are referred to as *class expressions*. Roles are referred to as *properties*. OWL differentiates between properties that connect individuals to individuals (object properties) from properties that connect individuals to literal values (data properties). An individual is uniquely referenced in an OWL

ontology using an international resource identifier (IRI) [36].

An OWL ontology can be represented using several different syntaxes. Since OWL is intended for use with the Semantic Web, most of these syntaxes are designed to be easily parsed by machines rather than to be read by humans. Table 2.2 compares the human-readable functional-style syntax of OWL [37] to the DL syntax commonly used in the literature.

Table 2.2: OWL functional-style syntax [37] vs. DL syntax. CE denotes class expressions, OPE denotes object property expressions, and I denotes individuals.

OWL syntax	DL syntax
ObjectIntersectionOf(CE_1 ... CE_n)	$C \sqcap D$
ObjectComplementOf(CE)	$\neg C$
ObjectSomeValuesFrom(OPE CE)	$\exists r.C$
SubClassOf(CE_1 CE_2)	$C \sqsubseteq D$
EquivalentClasses(CE_1 ... CE_n)	$C \equiv D$
ClassAssertion(CE I)	$a : C$
ObjectPropertyAssertion(OPE I_1 I_2)	$(a, b) : C$

An OWL ontology can be interpreted under a "direct" semantics [38] that aligns with the very expressive, yet still decidable, DL $\mathcal{SROIQ}$ [39] which extends $\mathcal{ALC}$ with transitive roles, a role hierarchy, nominals, inverse roles, and qualified cardinality restrictions. Inferences regarding an ontology can be computed using reasoners that employ tableau algorithms.

Aside from the full language, there are also several profiles [40] of OWL that are essentially sublanguages. Each profile restricts the use of some features in order to increase the tractibility of reasoning at the expense of expressivity. OWL EL is intended for ontologies that contain very large numbers of classes and/or properties. OWL QL is intended for ontologies use large volumes of instance data. OWL RL is a rule subset of OWL.

2.5 Works related to OwlLib

The task of integrating logic programs with DL reasoning is not original to this thesis. There has been great interest in producing useful integrations of rules and DLs. Much of the work is dedicated to producing an integration that retains decidability. While decidability is a desirable property, all existing attempts have required some restriction on expressivity to maintain it.

This section outlines important related works on integrating rules (sometimes more specifically LP) with DLs. The works listed here sorted in increasing order based on relavance and likeness to the approach taken by OwlLib. Section 2.5.1 and section 2.5.2 describe integrations that merge rules and DLs into a single formalism. These are introduced to contrast them with the approaches described in section 2.5.3, section 2.5.4, and section 2.5.5 which keep the DL ontology separate from the rule base, much like OwlLib, but do so with limited rule formalisms that are different from Prolog. The work outlined in section 2.5.5 is of particular interest due to a useful feature (the bi-directional flow of knowledge between the rules and ontology components). The works described in section 2.5.6 and section 2.5.7 both integrate Prolog with OWL (rather than some other DL). The work described in section 2.5.8 uses an approach very much like that of OwlLib, but possess subtle differences, one of which is its incompatibility with Android.

2.5.1 Description logic programs

Description logic programs (DLP) [7] are defined by the intersection of DLs and LP. In particular, the intersection is defined by the fragment of first-order logic that is encompassd by both DLs and LP called *def-Horn*, which is definite equality free Horn logic. The focus on the intersection allows for rules to be translated to axioms and axioms to be translated into rules.

2.5.2 Semantic Web Rule Language

The Semantic Web Rules Language (SWRL) [8, 9] is based on a combination of OWL with the Rule Markup Language (RuleML) [41] that extends the set of OWL axioms to include Horn-like rules. A SWRL rule consists of an antecedant and a consequent, each composed of a conjunction of atoms of the form *C(X)*, *P(x, y)*, *sameAx(x, y)*, *differentFrom(x, y)*, where C is an OWL description, P is an OWL property, and x, y are either variables, OWL individuals, or OWL data values. SWRL is known to be undecidable.

2.5.3 $\mathcal{AL}$-log

$\mathcal{AL}$-log [5] is a knowledge representation system based on the integration of the the DL $\mathcal{ALC}$ [31] and Datalog. Knowledge in $\mathcal{AL}$-log is maintained in two different subsystems called the structural subsystem and the relational subsystem. The structural subsystem is an $\mathcal{ALC}$ ontology consisting of a TBox and an ABox that only contains concept assertions, but $\mathcal{AL}$-log makes the unique name assumption regarding individuals. The relationional subsystem consists of *constrained* Datalog rules in which constraints are terms representing concept assertions from the structural subsystem. Query-answering for $\mathcal{AL}$-log is decidable [5].

2.5.4 CARIN

CARIN [6] combines function-free Horn rules with an $\mathcal{ALCNR}$ ($\mathcal{ALC}$ with unqualified cardinality restrictions and a role hierarchy) ontology. Rules in CARIN can contain concept assertions like $\mathcal{AL}$-log but also allows role assertions. CARIN also makes the unique name assumption for both components. Sound and complete reasoning procedures for CARIN-$\mathcal{ALCNR}$ are provided in [6]. The authors show that reasoning is decidable for non-recursive Horn rules. Unrestricted recursive Horn rules in CARIN-$\mathcal{ALCNR}$ knowledge bases lead to undecidability. This can be remedied by requiring that the rules be *role safe*, which requires

at least one variable from every role atom appearing in the body to also appear in an atom of a base predicate in the body.

2.5.5 dl-programs

dl-programs [11, 14, 15] provide an interface between a DL ontology and an extended logic program interpreted under the answer-set semantics [29]. It allows queries to be made to a DL ontology using a dl-query which can be a concept inclusion axiom (or its negation), a positive or negative concept assertion, or a positive or negative role assertion. A dl-atom is a structure of the form $DL[S_1\ op_1\ p_1,\ \ldots,\ S_i,\ op_i,\ p_i;\ Q](t)$, where each S_i is a concept or a role, each op_i is one of $\uplus$ or $\ominus$. $S_i \uplus p_i$ indicates that results from the query p_i should be included in the ontology as instances of S_i when answering the query Q. Inversely, $S_i \ominus p_i$ indicates that results from the query p_i should be included in the ontology as instances of $\neg S_i$ when answering the query Q.

HEX-programs [42] are a generalization of dl-programs that also permit access to external data sources other than DL ontologies. As with many other attempts to integrate DLs with logic programs, dl-programs aim to increase the expressivity of DLs by integrating them with rule formalisms such as logic programming and focus on maintaining decidability of the two formalisms by using Datalog as the logic programming formalism.

The feature of dl-programs that is of most interest to this thesis is the bi-directional flow of knowledge between a DL ontology and a logic program. Most related works mentioned thus far allow for knowledge expressed in ontologies to be accessed from rules but dl-programs allow for the knowledge expressed in a logic program to also contribute to the consequences of an ontology.

2.5.6 DR-Prolog

DR-Prolog [43] is a Prolog system for defeasible reasoning with rules and OWL ontologies. It is based on a translation of defeasible logic [44] into logic programs under the well-founded semantics [30]. It facilitates reasoning of OWL ontologies by translating OWL axioms into rules. Its implementation requires XSB[4] (used as the reasoning engine for the system) and SWI-Prolog[5] (used for translating OWL axioms into rules), neither of which are compatible with Android.

The authors of [43] argue that the advantage of coupling a non-monotonic formalism such as Prolog with a monotonic formalism such as OWL is that the non-monotonic formalism would be capable of representing dynamic knowledge that changed frequently while the monotonic formalism would be better suited for representing static truths that rarely changed. This view further motivates the integration of Prolog programs and OWL ontologies.

2.5.7 Thea

Thea [45] is a Prolog library for managing OWL 2 ontologies. It leverages the SWI-Prolog *semweb*[6] module to read and parse the ontologies. It uses an "axiom-oriented" representation in which predicates look like OWL constructs in the OWL 2 functional-style syntax [37] as opposed to the triple-based representation used by the SWI-Prolog *semweb* library.

Reasoning in Thea is performed by either rewriting the ontology as a logic program (based on DLP [7]) or by using the OWL RL [40] reasoning rules. The documentation hints at a plain Prolog reasoner that uses backward-chaining rules, but it appears to be removed from the public source code repository[7]. Thea is able to access external reasoners through

[4] `http://xsb.sourceforge.net/`

[5] `http://www.swi-prolog.org/`

[6] `http://www.swi-prolog.org/pldoc/doc_for\?object=section\%28\%27packages/semweb.html\%27\%29`

[7] The "Reasoning_using_Thea.txt" file in source code repository at `https://github.com/vangelisv/`

the OWL API but uses SWI-Prolog's bidirectional Prolog/Java *JPL*[8] module to do so, and thus is incompatible with Android.

The authors of [45] state that Prolog offers many advantages as a host programming language for working with ontologies due to its declarative features and pattern-matching styles of programming. The feature of Thea that is of most interest to this thesis is the decision to use Prolog predicates and terms that correspond to the functional-style syntax of OWL. This allows for the predicates to look much like their equivalent OWL constructs so that placing a query to an OWL reasoner or representing a complex class expression looks and feels natural. OwlLib adopts this same syntax for predicates that place queries to OWL reasoners.

2.5.8 Ontological Logic Programming

Ontological Logic Programming (OLP) [46] combines Prolog with DL reasoning by enabling the use of entities from an OWL ontology, such as individuals, classes, and properties, from within a Prolog program. The interpretation of terms from the ontology is delegated to an external ontology reasoner during the interpretation of the program. A simple meta-interpreter is used to invoke an OWL reasoner (Pellet [47]) when an ontological predicate, such as `ex:'Person'(X)`, is encountered. OLP keeps the OWL ontology completely seperate from the Prolog rule base. The implementation[9] of OLP leverages many libraries that are not compatible with Android.

The authors of [46] cite several advantages of OLP. Of these advantages, the two of most interest to this thesis are (1) the transparent use of DL reasoning from within logic programs and (2) the reuse of domain knowledge specified in ontologies. Knowledge defined in DL ontologies is intended to be interpreted in a certain fashion. The delegation of ontological

`thea` mentions a file called "owl2_basic_reasoner.pl" that implements these rules, but it is not present.

[8] `http://www.swi-prolog.org/packages/jpl/`

[9] http://sourceforge.net/projects/olp-api/

queries to external reasoners allows them to be answered in a decidable and pragmatic manner. Furthermore, the reuse of domain knowledge specified in ontologies can allow logic programs with access to DL reasoning to collaborate with other systems that make the same ontological commitments by leveraging the same ontologies. Like OLP, OwlLib also employs external OWL reasoners in order to offer these same advantages. However, OwlLib is fully compatible with Android and uses a different syntax.

2.6 Works related to SensorLib

Context-aware systems adapt their operations without explicit user intervention by taking context into account [18]. One of the initial requirements for context-awareness is a mechanism for obtaining context [48]. SensorLib is a Prolog library purposed for the aquisition of sensor observations, which constitute context, and making them available to Prolog programs. Once the necessary low-level information is aquired, it can be used at a higher level to perform tasks such as physical activity recognition.

Physical activity recognition is the process of detecting the activity a person is currently performing, such as standing, walking, or jogging. Various approaches have been applied, from mining observation data reported by body-worn sensors [49] and sensors attached to objects in the environment [50] to video recognition of activities [51]. Body-worn sensors can be obtrusive as they require sensors to be physically attached to body of the subject while object sensors can be expensive to deploy. An alternative to the above approaches is to use the on-board sensors available on mobile devices. In addition to being equipped with a number of sensors capable of measuring a myriad of physical properties, smartphones and other pocket-sized mobile devices make good platforms for activity recognition due to their ubiquity and low/none installation cost [52]. Mobile devices have successfully been used to perform activity recogntion for various purposes such as detecting the transportation mode

of a user (walking, biking, driving, etc.) [53, 54], estimation of energy expenditure [55], and detecting falls suffered by elderly people [56].

Chapter 3

OwlLib – A Prolog library for OWL

3.1 Introduction

OwlLib is a Prolog library for querying and manipulating OWL ontologies from within Prolog programs on Android devices. It provides terms for representing OWL constructs and predicates for managing OWL ontologies and OWL reasoners, executing low-level and high-level queries of the ontologies using the reasoners, and manipulating OWL ontologies. It is designed specifically for the JPR Prolog interpreter due to JPR's compatibility with Android. Some of the predicates of OwlLib are implemented entirely in Java while the implementation of others is split between Java and Prolog source code.

OwlLib leverages the OWL API [20] to load, parse, and provide a low-level representation of OWL ontologies and constructs. The OWL API also provides a common interface for accessing various implementations of OWL reasoners. Android-compatible reasoners that implement the interface are queried by OwlLib predicates to compute inferences regarding OWL ontologies. The use of the OWL API could potentially be replaced with another OWL framework such as Jena [57], though this would require significant retooling of certain components of the library. Though it primarily targets the Android platform, OwlLib is

intended to be compatible with any platform that possesses adequate memory to support a Java Virtual Machine and the data contained in the Prolog knowledge base and OWL ontologies loaded by the library.

This chapter discusses the design and implementation of OwlLib and is organized as follows. Section 3.2 presents the structure of OwlLib and describes the interaction of its components. The ontology and reasoner management functionality of OwlLib is outlined in section 3.3. Details regarding the Prolog representation of OWL entities are discussed section 3.4 as well as the manner is which complex descriptions can be created from them. Section 3.5 is dedicated to explaining how OWL reasoners are queried using OwlLib predicates and how the results of the queries are made available to a Prolog program. The implementation of a special predicate that allows for the bi-directional flow of knowledge between an OWL ontology and a Prolog program is outlined in section 3.6. The facilities provided by OwlLib for manipulating an ontology are outlined in section 3.7. Lastly, section 3.8 discusses how the issues regarding the orthogonal semantics of Prolog and OWL are handled in OwlLib. A full listing of the predicates provided by OwlLib can be found in appendix A.

3.2 Structure of OwlLib

OwlLib is developed for the JPR Prolog environment and consists of many terms and predicates for working with OWL ontologies and reasoners. While some functionality of the library is implemented in Prolog, the majority is implemented in Java to interact with Java-based tools, namely the OWL API and the external OWL reasoners. The general structure of OwlLib is illustrated in figure 3.1.

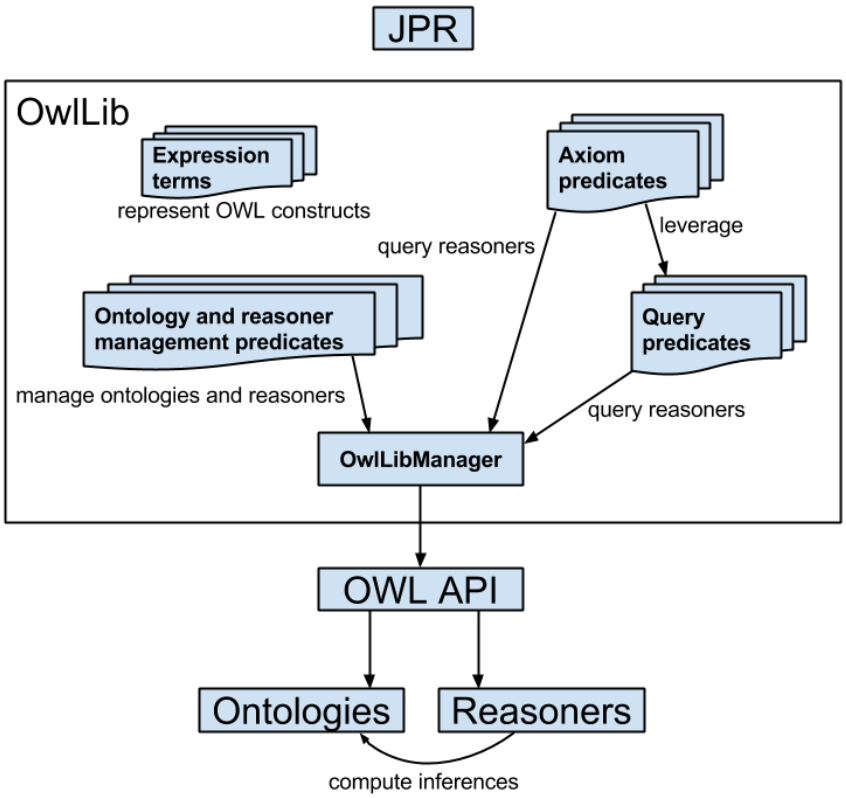

Figure 3.1: The relationship between OwlLib and third-party tools. The Java implementations of OwlLib predicates and terms interact with the OWL API and external ontologies and reasoners through an instance of the `OwlLibManager` Java class.

The OWL API [20] is an Application Programming Interface (API) for working with OWL ontologies in Java. It provides an in-memory Java representation of OWL ontologies and provides functionality for loading and parsing of OWL ontologies. The OWL API takes an "axiom-centric" view of ontologies in which an ontology is treated as a set of axioms. The OWL API also provides a Java representation of OWL constructs such as entities, expressions, and axioms. In addition, the OWL API defines the Java interface `OWLReasoner` for accessing OWL reasoners in a uniform way from Java.

The Java class `OwlLibManager` provides backend functionality by acting as a bridge between OwlLib predicates and terms and the OWL API. It stores and manages loaded ontologies and the reasoners associated with them. It also provides functionality for obtaining

the Java representations of OWL constructs provided by the OWL API. There is only one instance of `OwlLibManager` that is created when OwlLib is loaded into JPR. A reference to this instance is provided to each instance of each Java-based OwlLib predicate and term as a means to access the reasoners.

Each predicate and term of OwlLib (with the exception of `dl/2`, discussed in section 3.6, and part of the axiom predicates that accept unbound arguments, discussed in section 3.5) is implemented in Java in order to access the functionality provided by `OwlLibManager`. The Prolog implementations of the predicate `dl/2` and the rules defining the functionality of axiom predicates that accept unbound arguments reside in a Prolog file named `owllib.pl`.

3.3 Ontology and reasoner management

The ontology management predicates of OwlLib provide functionality for the loading, managing, and saving of OWL ontologies while the reasoner management predicates are provided for the creation and management of OWL reasoners associated with loaded ontologies. Loaded ontologies and reasoners are referenced from Prolog using atoms that serve as "handles" that are defined at the time the ontology or reasoner is loaded. Some of the ontology and reasoner management predicates of OwlLib are listed in table 3.1 while the rest are listed in appendix A.

Table 3.1: Ontology and reasoner management predicates

Predicate	Functionality
`createOntology(+O)`	Creates an empty ontology with the handle `O`
`loadOntology(+Path, +O)`	Loads an ontology with the handle `O` from a file on the path `Path` of the file system
`useOntology(+O)`	Sets the ontology with the handle `O` to be the active ontology
`createReasoner(+R, +Type, +O)`	Creates an reasoner with the handle `R` of the type `Type` for the ontology with the handle `O`
`saveOntology(+Path, +O)`	Saves the ontology with the handle `O` to a file with the path `Path` on the file system
`unloadOntology(+O)`	Unloads the ontology with the handle `O`

OwlLib allows for multiple OWL ontologies to be loaded at any given time. An ontology can be loaded from a text file or a blank ontology can be created from scratch. Of the loaded ontologies, one is set to be the "active" ontology. Each loaded ontology can have multiple reasoners associated with it. Of the reasoners associated with a particular ontology, one is set to be the "active" reasoner for that ontology. The Java implementations of axiom predicates and query predicates submit queries to the active reasoner of the active ontology as shown in figure 3.2.

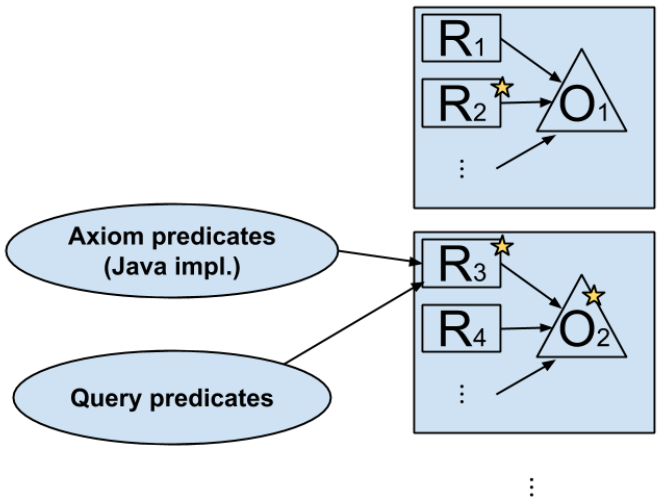

Figure 3.2: Active ontologies and reasoners. The ontology O_2 is set to be the active ontology and the reasoner R_3 is set to be the active reasoner of O_2, as denoted by the stars. The Java implementations of axiom predicates and query predicates submit queries to the active reasoner of the active ontology.

An example of how to use the ontology and reasoner management predicates is demonstrated below.

```
?- loadOntology('/path/to/ontol1/o1.owl', o1).          % (1)
?- createReasoner(r1, jfact, o1).                       % (2)
?- createOntology(o2).                                  % (3)
?- useOntology(o1).                                     % (4)
?- classAssertion(ex:Person, ex:Bob),                   % (5)
   assertAxiom(classAssertion(ex2:Human, ex:Bob), o2).
?- saveOntology('/path/to/ontol2/o2.owl', o2).          % (6)
```

Query (1) loads the ontology stored in a file named **o1.owl** and sets the handle **o1** as the reference for it. Query (2) creates a JFact [21] reasoner with the handle **r1** for the ontology **o1**. The reasoner **r1** is automatically set to be the active reasoner for **o1**. Query (3) creates an empty ontology with the handle **o2** and query (4) manually sets the active ontology to be **o1** again. Query (5) submits a query to the active reasoner **r1** regarding the entailment of an axiom by **o1** and asserts a similar axiom into ontology **o2** (**ex:Person**, **ex:Bob**, **ex2:Human**, and **ex:Bob** are all atoms, as explained in section 3.4.1). Query (6) saves ontology **o2** to a file named **o2.owl**.

OwlLib supports the reasoners JFact [21] (a Java port of the OWL DL reasoner FaCT++ [58]) and ELK [22] (a reasoner tailored towards $\mathcal{EL}$ ontologies) due to their compatibility with Android and their implementation of the OWL API **OWLReasoner** interface. Efforts outlined in [59] and [60] were made to determine what other existing OWL reasoners were compatible with Android. The investigation found that jcel [61], an EL+ reasoner, was compatible by default while Pellet [47], HermiT [62], and CB [63] required extensive modification in order to function on the Android platform.

In addition to the supported reasoners, OwlLib also provides a mechanism for adding unsupported reasoners that implement the **OWLReasoner** interface of the OWL API. However, this requires for the reasoner to be constructed manually in Java before supplying to OwlLib for use.

```
// obtain the instance of OwlLibManager from OwlLib
OwlLibManager olm = owllib.getManager();
// obtain a reference to the active ontology from the OwlLibManager
OWLOntology o = olm.getActiveOntology();
// manually create an instance of the new reasoner using the ontology object
OWLReasoner r = new FutureReasonerFactory().createReasoner(o);
// supply OwlLib with the created reasoner to be associated
//    with the active ontology using the handle "future"
olm.createReasoner("future", r, jpr.termFactory());
```

3.4 Representing OWL in Prolog

3.4.1 Representing basic entities

Classes, individuals, and properties are the most basic entities in an OWL ontology's domain. Since these entities are represented in OWL using IRIs, atoms corresponding to their IRIs are used to represent them in OwlLib. For example, the atom '`http://www.example.com/owl/stuff#Bob`' might be used to represent an individual with the name "Bob". Note that atoms representing full IRIs must be quoted.

OWL allows for IRIs to be abbreviated using prefix names. Likewise, OwlLib allows for IRI atoms to be abbreviated using the **`addPrefix/2`** predicate. The evaluation of:

```
?- addPrefix(ex, 'http://www.example.com/owl/stuff#').
```

allows the prefix name **`ex:`** to be used in place of the IRI prefix '`http://www.example.com/owl/stuff#`'. The atom '`http://www.example.com/owl/stuff#Bob`' could then be abbreviated to simply **`ex:Bob`**. The abbreviated atom does not require quotes. OwlLib predicates and terms will expand such an abbreviated atom into its full IRI when it is received as an argument.

3.4.2 Representing complex descriptions

Expressions in OWL are used to create complex descriptions from basic entities. Expressions are supported by OwlLib through expression terms. Each expression term corresponds in syntax to its OWL functional-style [37] counterpart. The name and arity of each expression term matches its counterpart, with the exception that the terms begin with a lower-case letter. The expression terms corresponding to OWL expressions that accept a variadic number of arguments accept a single Prolog list containing the arguments. This syntax is a natural Prolog representation of OWL constructs and is inspired by that of Thea [45]. For

example, the expression term

```
objectIntersectionOf([ex:Parent, ex:Male])
```

corresponds to the OWL class expression

ObjectIntersectionOf(ex:Parent ex:Male)

representing the set of individuals belonging to the classes **Parent** and **Male**. Some of the implemented expression terms are listed in table 3.2 while the rest are listed in appendix A.

Table 3.2: Class expression terms

Term	Representation
`objectIntersectionOf(+CEs)`	Represents the set of individuals that are instances of every class expression in the Prolog list **CEs**
`objectUnionOf(+CEs))`	Represents the set of individuals that are instances of at least one of the class expressions in the Prolog list **CEs**
`objectComplementOf(+CE)`	Represents the set of individuals that are not instances of the class expression **CE**
`objectSomeValuesFrom(+OPE, +CE)`	Represents the set of individuals that are connected by **OPE** to an individual that is an instance of **CE**
`objectAllValuesFrom(+OPE, +CE)`	Represents the set of individuals that are connected by **OPE** only to individuals that are instances of **CE**
`objectMinCardinality(+N, +OPE, +CE)`	Represents the set of individuals that are connected by **OPE** to at least **N** different individuals that are instances of **CE**
`objectMaxCardinality(+N, +OPE, +CE)`	Represents the set of individuals that are connected by **OPE** to at most **N** different individuals that are instances of **CE**

3.5 Querying OWL reasoners

3.5.1 Axiom predicates for querying OWL reasoners

Axiom predicates are the primary means of querying OWL reasoners using OwlLib. Similar to expression terms, each axiom predicate possesses a name and arity identical to its OWL functional-style [37] counterpart with the exceptions that its functor begins with lower-case letter. When an axiom predicate is evalutated, a query is made to the active reasoner of the active ontology to either determine if the axiom represented by the predicate is entailed by

the ontology or to bind any unbound arguments based on the consequences of the ontology. Some of the axiom predicates of OwlLib are listed in table 3.3 while the rest are listed in appendix A.

Table 3.3: Axiom predicates

Predicate	Representation
`subClassOf(?CE1, ?CE2)`	States `CE1` is a subsumed by `CE2`
`equivalentClasses(+CEs)`	States that class expressions in the Prolog list `CEs` are equivalent
`disjointClasses(+CEs)`	States that class expressions in the Prolog list `CEs` are pairwise disjoint
`classAssertion(?CE, ?I)`	States that `I` is an instance of `CE`
`sameIndividual(+Is)`	States that individuals in Prolog list `Is` are the same individual
`differentIndividuals(+Is)`	States that individuals in Prolog list `Is` are all different individuals
`objectPropertyAssertion(+OPE, +I1, ?I2)`	States `I1` is connected by `OPE` to `I2`
`negativeObjectPropertyAssertion(+OPE, +I1, +I2)`	States `I1` is not connected by `OPE` to `I2`

The type of query that an axiom predicate makes and the manner in which the query is carried out depends on the instantiation of the predicate's arguments. If the predicate contains no unbound variables, the Java implementation queries the active reasoner of the active ontology to determine if the ontology entails the represented axiom. If it does, then the axiom predicate evaluates to true; otherwise, it evaluates to false. For example, assuming that **`ex:Father`** and **`ex:Person`** are atoms representing the OWL classes Father and Person respectively, the following query:

```
?- subClassOf(ex:Father, ex:Person).
```

would evaluate to true if the reasoner is able to infer that ontology entails that the class Father is a subclass of Person. Similarly, the query:

```
?- classAssertion(ex:Father, ex:Bob).
```

would evaluate to true if the reasoner is able to infer that the ontology entails that Bob is an instance of the class Father. This type of query is considered "low-level" as the Java

implementation of the predicate queries the reasoner directly, as illustrated in figure 3.3.

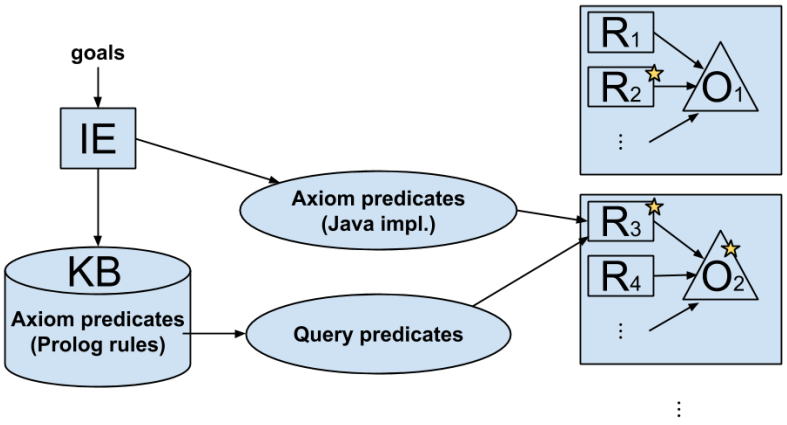

Figure 3.3: Query delegation in OwlLib. The Java implementations of axiom predicates query the reasoners directly while the axiom predicate rules do so using query predicates.

The treatment of an axiom predicate that contains an unbound variable is handled using a Prolog rule (these rules are discussed in section 3.5.2 in relation to query predicates also defined in the section). When an axiom predicate containing an unbound variable is evaluated, OwlLib queries the reasoner. However, since the axiom represented by the predicate is not fully formed (it contains a variable), instead of quering for axiom entailment, a query for all the entities that satisfy the axiom is made. The query returns a set of results. Each result is bound, in turn, to the variable. For example, the following query is not checking axiom entailment but rather is ascertaining all the individuals inferred to be instances of Person:

```
?- classAssertion(ex:Person, X).
X = ex:Bob ;
X = ex:Mary ;
...
```

As demonstrated by this example, `X` can be repeatedly bound to different results from the query. A query of this type is considered "high-level" as it is implemented by a Prolog rule

that leverages a query predicate (discussed in section 3.5.2) that accesses a reasoner at a "low-level" in Java, as illustrated in figure 3.3.

3.5.2 Query predicates for low-level querying of OWL reasoners

In order to implement the queries issued by axiom predicates that contain unbound variables, OwlLib leverages a number of low-level query predicates that are used to submit particular queries to the OWL reasoner. All of the query predicates are implemented entirely in Java and each is responsible for a single query. A query predicate expects one or more input arguments to use as parameters for the query and an unbound variable to as the last argument to be used for output. Some of the query predicates of OwlLib are listed in table 3.4 while the rest are listed in appendix A.

Table 3.4: Query predicates

Predicate	Functionality
`getSubClasses(+CE, -Iter)`	Gets set of named subclasses of class expression `CE` and binds iterator of set to `Iter`
`getSuperClasses(+CE, -Iter)`	Gets set of named superclasses of class expression `CE` and binds iterator of set to `Iter`
`getEquivalentClasses(+CE, -Iter)`	Gets set of named equivalent classes of class expression `CE` and binds iterator of set to `Iter`
`getDisjointClasses(+CE, -Iter)`	Gets set of named disjoint classes of class expression `CE` and binds iterator of set to `Iter`
`getSameIndividuals(+I, -Iter)`	Gets set of individuals that are the same as `I` and binds iterator of set to `Iter`
`getDifferentIndividuals(+I, -Iter)`	Gets set of individuals that are different than `I` and binds iterator of set to `Iter`
`getTypes(+I, -Iter)`	Gets set of named classes of which `I` belongs and binds iterator of set to `Iter`
`getInstances(+CE, -Iter)`	Gets set of individuals that are instances of class expression `CE` and binds iterator of set to `Iter`
`getObjectPropertyValues(+I, +OPE, -Iter)`	Gets set of individuals that are connected to `I` by `OPE` and binds iterator of set to `Iter`

As demonstrated in section 3.5.1, the axiom predicate **`classAssertion/2`** can be used to obtain all the instances of a given class. The high-level is implemented by the following rule:

```
classAssertion(CE, I) :-
    % if a class expression is given and an instance is expected,
    nonvar(CE), var(I),
    % then get the set of instances from the reasoner...
    getInstances(CE, Iter),
    % ...and bind the first (if any) to I
    next(Iter, R), owlObjToTerm(R, I).
```

The predicate **getInstances/2** is used to obtain all of the instances of a given class expression as inferred by the reasoner. The reasoner returns the instances as result set and the query predicate binds its output argument **Iter** to a JPR **ReferenceTerm** containing an iterator to the set. The iterator can then be used to access every element in the set via **next/2**, described in section 3.5.3.

3.5.3 Recursion and backtracking with iterators

The evaluation of a query predicate submits a query to the active reasoner which returns a result set. The query predicate then binds a JPR **ReferenceTerm** that contains an iterator to the set to its output argument. The predicate **next/2** can then be used to access each element of the set.

Every time that **next/2** is evaluated, it obtains the next element of the iterator and places it in a **ReferenceTerm** that is then bound to its second argument as output. Thus it is possible to access all the elements in the result set by repeatedly passing the same iterator to **next/2**. In the case of OwlLib, the elements are entities of an OWL ontology represented in Java by classes of the OWL API and can be converted to a Prolog representation using **owlObjToTerm/2**:

```
?- getInstances(CE, Iter),
   next(Iter, R),      % bind the next element to R using a reference term
   owlObjToTerm(R, I). % convert the reference term R to a Prolog atom I
```

The evaluation of **next/2** also creates a backtrack point if the iterator has not accessed all the elements in the result set. Upon backtracking, the next element of the iterator is bound

and another backtrack point is created if the iterator still has more elements to visit. This process continues until the iterator has visited all of the elements in the result set. For example, reconsider the following rule defining part of the functionality of **classAssertion/2**:

```
classAssertion(CE, I) :-
    nonvar(CE), var(I),
    getInstances(CE, Iter),
    next(Iter, R),            % backtrack point created if Iter has more elements
    owlObjToTerm(R, I).
```

When **classAssertion/2** is evaluated, only a single query to the reasoner is made when **getInstances/2** is evaluated. However, multiple backtrack points may be created depending on the number of elements still unvisisted by the iterator of the result set.

3.6 Bi-directional flow of knowledge

The axiom predicates and query predicates of OwlLib allow for queries to an OWL reasoner to be made from within a Prolog program. However, both types of predicates only allow knowledge to flow in one direction, from the ontology to the program. OwlLib also provides a special predicate **dl/2** to allow the consequences of a Prolog program to be added to the knowledge expressed by an OWL ontology and thus possibly be a part of the inferences computed by an OWL reasoner. **dl/2** is inspired by the work of Eiter et al. on dl-programs [11, 14, 15].

The syntax of **dl/2** is

$$\texttt{dl([}E_1\ op_1\ p_1\texttt{/}N\texttt{, ..., }E_n\ op_n\ p_n\texttt{/}N\texttt{], Axiom)}$$

where each combination $E_i\ op_i\ p_i/N$ is called an "increaser" and Axiom is a term corresponding to class assertion or object property assertion. An increaser contains a class expression or object property expression E_i, an indicator operator op_i (either **+=** or **-=**), and the name of a unary or binary predicate and the value of its arity p_i/N.

When an instance of **dl/2** is evaluated, the knowledge derived from consequences of a Prolog program is first temporarily added to the active ontology. The query expressed by the axiom predicate argument is then submitted to the active reasoner. Lastly, the temporary knowledge is removed, returning the ontology to its previous state.

An increaser containing a class expression **E** and the indicator operator **+=** causes the results from calling the unary predicate with functor **p** to be temporarily added to the OWL ontology as instances of the class expression. When the reasoner answers the query represented by **Axiom**, it can take the new temporary knowledge into consideration when computing inferences. For example, consider an ontology containing only the following axiom stating that every Father is also a Parent and Male:

```
# every Father is also a Parent and Male
SubClassOf(Father ObjectIntersectionOf(Parent Male))
```

and a Prolog program only containing the following fact stating that **bob** is a **father**:

```
% bob is a father
father(bob).
```

Note that since the ontology and Prolog knowledge base are kept separate by OwlLib, the ontology does not imply that **bob** is a Father nor does the program imply that **bob** is a Parent or Male. However, the following use of **dl/2** evaluates to true because every consequent of **father/1** is temporarily asserted to be an instance of Father before querying the reasoner to ascertain if **bob** would be classified as a Parent by the ontology:

```
% If every father from the program is also a Father in the ontology,
%   then is 'bob' from the program also classified as a Parent
%   by the ontology?
?- dl(['Father' += father/1], classAssertion('Parent', bob)).
true.
```

Conversely, if the indicator operator **-=** is used, then the results are temporarily added as instances of the *complement* of the class expression:

```
% fact from program stating "'fido' is a animal"
animal(fido).

% If every animal from the program is not a Person in the ontology,
%   then is 'fido' from the program classified as a Person
%   by the ontology?
?- dl(['Person' -= dog/1], classAssertion('Person', fido)).
false.
```

Similarly, if the `+=` indicator operator is used with a property expression then the results from calling the supplied binary predicate are temporarily added to the ontology as being connected by the property expression and if `-=` is used they are temporarily added as explicitly *not* being connected.

In order to achieve this functionality, `dl/2` first collects the results of calling the predicate associated with each increaser. Axioms corresponding to the results are then asserted into the ontology. Next, the active reasoner is used to determine if the ontology is still consistent. If the ontology is found to be inconsistent, the added axioms are removed and `dl/2` fails. If the ontology remains consistent, the corresponding axiom predicate is evaluated, placing a query to the reasoner as usual. After the evaluation of the axiom predicate completes, the axioms that were temporarily added to the ontology are removed.

3.7 Ontology manipulation

In addition to allowing OWL reasoners to be queried, OwlLib also provides ontology manipulation predicates for manipulating OWL ontologies from Prolog programs. The ontology manipulation predicates are listed in table 3.5.

Table 3.5: Ontology manipulation predicates

Predicate	Functionality
`assertAxiom(+A, +O)`	Adds the axiom represented by `A` into the ontology with handle `O`
`retractAxiom(+A, +O)`	Removes the axiom represented by `A` from the ontology with handle `O`
`isConsistent(+O)`	Determines if the ontology with handle `O` is consistent

Axioms can be added to an ontology using **assertAxiom/2**. It should be noted that OwlLib does not prevent contradictory axioms from being added to an ontology. If contradictory axioms are added to an ontology, the ontology will become inconsistent. When an ontology is inconsistent, its reasoners will not answer any queries regarding the ontology but will throw an exception instead. The predicate **isConsistent/1** can be used to determine whether an ontology is consistent after an axiom has been added to it. If the addition of an axiom does lead to an ontology inconsistency, it can be removed using **retractAxiom/2** to return the ontology to a consistent state.

```
% assert that 'bob' is a Person
?- assertAxiom(classAssertion(ex:Person, bob)).
true.

% assert that 'bob' is not a Person only if it does not
%   make the ontology inconsistent
?- assertAxiom(classAssertion(objectComplementOf(ex:Person), bob)),
   \+ isConsistent(ontol1),
   retractAxiom(classAssertion(objectComplementOf(ex:Person), bob)).
true.
```

3.8 Handling issues of integrating Prolog with OWL

The semantics of Prolog and OWL are not readily compatible with one another. Prolog is non-monotonic and makes the closed-world assumption (CWA) and the unique name assumption (UNA) while OWL is monotonic and makes the open-world assumption (OWA) but does *not* make the unique name assumption. OwlLib keeps the knowledge expressed an OWL ontology separate from that expressed in a Prolog program and leverages an OWL reasoner to compute the inferences regarding an ontology when axiom predicates and query predicates are evaluated by the JPR inference engine. OwlLib is true to the semantics of each component when each is considered in isolation though no unifying global semantics are given. However, it is arguable whether such a unifying global semantics would be beneficial

in most practical applications. In the following sections, relevant differences between Prolog and OWL are addressed and issues that they might cause in applications are highlighted.

3.8.1 Non-monotonicity vs. monotonicity

In OwlLib, an OWL reasoner computes inferences based on the contents of an OWL ontology. It has no access to the Prolog KB so the contents of the Prolog KB have no effect on the results obtained by the reasoner. Furthermore, axioms asserted into an OWL ontology cannot reduce the inferences computed by an OWL reasoner (however they can cause the ontology to become inconsistent). Therefore, monotonicity is preserved on the OWL side of things in OwlLib, in the sense that one may query and manipulate the OWL ontology in the usual OWL way.

Non-monotonicity is preserved within the Prolog knowledge base as it always is, through negation-as-failure. Considering the following canonical example, assuming that the KB contains no **`guilty(bob)`** facts, **`bob`** is considered innocent:

```
innocent(X) :- \+ guilty(X).

?- innocent(bob).  % true until guilty(bob) is asserted
```

The above rule can be replaced with the following one that ascertains the innocence of an individual based on consequences of the ontology.

```
innocent(X) :- \+ classAssertion(ex:Guilty, X).

?- innocent(bob). % true until the axiom ClassAssertion(Guilty bob)
                    becomes a consequence of the ontology
```

Therefore, OwlLib enables open-world reasoning in Prolog by delegating the computation of inferences of the ontology to dedicated OWL reasoners.

3.8.2 Closed-world assumption vs. open-world assumption

The reasoning performed by the OWL reasoners is performed only over knowledge expressed in the ontology and in isolation from the Prolog program. Therefore, the OWA is able to be employed during the extent of the reasoning.

On the Prolog side of things, the inference engine can leverage the CWA as usual. Consider the following query ascertaining if **bob** is an instance of Person:

```
?- classAssertion(ex:Parent, bob).
```

If the represented class assertion axiom is entailed by the ontology, the query would return true. Otherwise, it would evaluate to false. However by the OWA, if it cannot be determined if **bob** is an instance of Person or an instance of the *complement* of Person, the membership of **bob** in the class Person is "unknown". The following query will evaluate to true if it is unknown whether or not **bob** is an instance of Person:

```
?- \+ classAssertion(ex:Person, bob),
   \+ classAssertion(objectComplementOf(ex:Person), bob).
```

Thus by submitting leveraging negation-as-failure while submitting both positive and negative queries to the reasoner, it can be determined whether certain consequences regarding an ontology are "unknown". For convenience, such queries could be defined as rules:

```
unknown(classAssertion(CE, I)) :-
  \+ classAssertion(CE, I),
  \+ classAssertion(objectComplementOf(CE), I).
```

3.8.3 The unique name assumption

Prolog makes the unique name assumption (UNA) which means that any terms that are not syntactically identical are considered to be different and thus not equal. For instance, the atoms **bob** and **bobby** are treated as different when compared:

```
?- bob == bobby.
false.
```

On the other hand, OWL does not make the unique name assumption. The same entity can be referenced using two different IRIs. The reasoning performed by the OWL reasoners take the equivalence of entities into consideration. If **bob** and **bobby** are asserted to be the same individual and **bob** is asserted to be a member of Person, then **bobby** will also be inferred as belonging to Person. Such is the case for any queries regarding any consequences derivable regarding **bob** will also be true of **bobby**. Thus OwlLib does not impose the UNA on the OWL ontology.

To align with Prolog's treatment of syntactically different atoms, both **bob** and **bobby** are returned for the query asking for all the instances of Person.

```
?- classAssertion(ex:Person, X).
X = bob ;
X = bobby ;
...
```

Therefore, any atoms used to represent entities from the ontology should be compared using the appropriate axiom predicate for testing (in)equality (such as **sameIndividuals/1**) rather than the Prolog comparison operators.

```
?- sameIndividuals([bob, bobby]).
true.
```

3.9 Conclusion

OwlLib is an integration of logic programming with description logic reasoning for mobile devices. It provides predicates and terms for working with OWL ontologies and reasoners. External tools are leveraged by OwlLib in order to ensure that the knowledge expressed in OWL ontologies is interpreted in manner in which it was intended. The OWL API is leveraged to load and parse OWL ontologies as well as provide an in-memory Java representation

of them and other OWL constructs. The `OWLReasoner` interface of the OWL API is used to provide uniform access for querying OWL reasoners that compute inferences regarding OWL ontologies. Though another Java OWL framework could potentially be used in place of the OWL API, it would have to provide an interface by which to access OWL reasoners. The separation of the Prolog and OWL reasoning systems allows for the two formalisms which possess different semantics to be used together in the same system.

Chapter 4

SensorLib – A Prolog library for accessing Android sensors

4.1 Introduction

Most Android devices come equipped with on-board sensors such as accelerometers, gyroscopes, light sensors, GPS, and sometimes even thermometers capable of observing a variety of events regarding the device's current state and the state of the environment around the device. The Android Software Development Kit (SDK) [23] provides a low-level framework consisting of several Java classes and interfaces for accessing the sensors and obtaining the observations they report. The SDK imposes some responsibilities on the developer such as the need to manually register and unregister the listeners that receive observations from sensors.

SensorLib is a Prolog library developed for the JPR Prolog environment that facilitates access to observations reported by sensors on Android devices from within Prolog programs. SensorLib handles the reported observations in an efficient and customizable manner using observation managers (OMs). An OM automatically receives, stores, and manages sensor

observations that are reported by an on-board Android sensor. An OM can be set to store only a set number of sensor events or all the sensor events that occur over a specified time duration. Observations are stored chronologically in a sliding window. When a window reaches its capacity, old observations are removed as new observations are added in a first-in first-out fashion. A clone, or snapshot, of the collection of observations managed by an OM can be obtained at any time. The observations stored in the clone can each be accessed in turn using an iterator. SensorLib is only compatible with Android 4.2 (API Level 17) and higher due to certain features of the SDK that it leverages.

This chapter discusses the design and implementation of SensorLib and is organized as follows. Section 4.2 discusses how sensor observations are obtained using the Android SDK. The structure of SensorLib is illustrated in section 4.3. Observations managers and the predicates provided for working with them are outlined in section 4.4. Section 4.5 details how to obtain a clone of the active window of observations maintained by an observation manager. Common statistical operations that are included with SensorLib are outlined in section 4.6. Lastly, section 4.7 provides details regarding the representation of observations in SensorLib. A full listing of the predicates provided by SensorLib can be found in appendix B.

4.2 Receiving sensor observations with the Android SDK

The Android SDK does not allow for a sensor to be polled directly for its most recent observation. Instead, the newest observation is delivered to listener objects whenever a sensor reports a new observation. The interface **`SensorEventListener`** declares the callback method **`onSensorChanged(SensorEvent)`** and is used to define listeners to receive observations reported by motion, position, and environmental sensors. When a sensor reports a new observation, the **`onSensorChanged(SensorEvent)`** method of all listeners registered

with that particular sensor is called. The **SensorEvent** object received as an argument contains the values of the observation, a timestamp indicating when the observation occurred, the estimated accuracy of the observation, and a reference to the sensor that reported the observation.

Similarly, the interface **LocationListener** declares the callback method **onLocationChanged(Location)** and is used to define listeners to receive observations reported by the GPS and network location providers regarding the current geographical location of a device. The **Location** object contains information similar to that of **SensorEvent** objects (the latitude and longitude of the observation, the time of the observation, etc.).

Though the Android SDK delineates between sensors and location providers, SensorLib provides a uniform view, classifying both as sensors that report observations.

4.3 Structure of SensorLib

SensorLib is developed for the JPR Prolog interpreter to facilitate access to observations reported by sensors on Android devices from within Prolog programs. As a Prolog library, SensorLib provides a number of predicates for checking the availability of sensor types, creating and managing observation managers, cloning windows of obervations, and accessing the obervation data stored in those windows. Each of the predicates is developed entirely in Java due to the low-level nature of the tasks performed. The general structure of SensorLib is illustrated in figure 4.1.

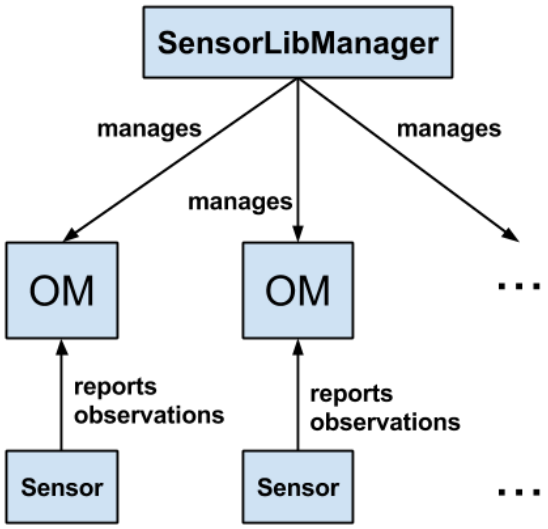

Figure 4.1: The structure of SensorLib

SensorLib provides observation managers (OMs) to automatically handle the acquisition, storage, and management of sensor observations. An OM is registered with a particular sensor and stores observations received from that sensor in a sliding window of size-based or time-based capacity. The window can be cloned and the observations stored in the cloned window can be obtained by iterating over the clone.

The class **`SensorLibManager`** is responsible for managing OMs. When the application loses focus (a new application is opened or the user navigates back to the device's home screen), **`SensorLibManager`** unregisters all OMs that have not been manually unregistered. When the application regains focus, all of the OMs are reregistered with their respective sensors automatically (unless they were unregistered manually). The automatic unregistering and reregistering of OMs relieves the developer from the burden of doing so manually. This default behaviour can be disabled if a developer wishes to leave the OMs registered so that they may receive observations even when an application is not in focus.

The library is loaded into JPR in the usual way except an Android **`Context`** is passed to the constructor:

```
InferenceEngine jpr = new InferenceEngine();
// ‘‘this’’ refers to the current Activity (which is a valid Context)
SensorLib sensorlib = new SensorLib(this);
jpr.termFactory().loadLibrary(sensorlib);
```

The **Context** is used by the instance of **SensorLibManager** in order to (1) obtain the system service managers for accessing sensors and location providers and (2) register callback methods that automatically register/unregister OMs when an application gains and loses focus.

By default, all code for an Android applications runs on the main UI thread. Time-intensive tasks need to be performed on different threads as to not block the UI thread. Most sensors are capable of reporting observations rapidly causing an OM to constantly ensure the capacity of its window is maintained. Therefore, **SensorLibManager** creates a background thread called the *OM thread* on which sensors report their observations to OMs. The use of the OM thread prevents the processes of rapidly receiving and storing observations and enforcing window capacity from blocking the UI thread.

Not all Android devices possess the same types of sensors. Therefore, SensorLib provides two predicates for determining what sensors are available on a device. The predicate **getSensorTypes(-Types)** provides a list of all available sensors while **hasSensor(+Type)** indicates if a particular sensor type is present:

```
?- getSensorTypes(Sensors).
Sensors = [accelerometer, gyroscope, lightSensor, ...]

?- hasSensor(acceleromter).
true.
```

Sensor types are represented by using the sensor type atoms listed in table 4.1.

Table 4.1: Sensor type atoms

`accelerometer` Accelerometer	`gameRotationVectorSensor` Game rotation vector
`gravitySensor` Gravity	`geomagneticRotationVectorSensor` Geomagnetic rotation vector
`gyroscope` Gyroscope	`magneticFieldSensor` Magnetic field
`linearAccelerationSensor` Linear acceleration	`proximitySensor` Proximity
`rotationVectorSensor` Rotation vector	
`ambientTemperatureSensor` Ambient temperature	`gps` GPS
`lightSensor` Light	`networkLocationSensor` Network location
`pressureSensor` Pressure	
`relativeHumiditySensor` Relative humidity	

4.4 Observation managers

Sensors report a stream of observations, therefore it is impractical and inefficient to attempt to store every observation reported. A popular technique in data stream management systems is to store a sliding window of data in which old data is neglected once new data becomes available. The advantage of such an approach is that recent data is emphasized over older data [19]. SensorLib supplies observation managers (OMs) to implement the sliding window technique for maintaining sensor observations. OMs provide the additional advantage of low memory consumption because only a limited number of observations are maintained in memory at a given time. The predicates for interacting with OMs and their related functionality can be found in table 4.2.

Table 4.2: Observation manager predicates

Predicate	Functionality
`createObsMgr(+H, +Type, +Rate)`	Creates an OM with handle `H` and registers it with sensor of type `Type` with requested sampling rate `Rate`
`setCapacity(+H, +CapType, +CapVal)`	Sets the capacity type of OM with handle `H` to `CapType` (either `size` or `time`); capacity set to value `CapVal` (either floating point number representing time or integer representing size)
`destroyObsMgr(+H)`	Unregisters OM with handle `H` and discards it completely
`unregisterObsMgr(+H)`	Unregisters OM with handle `H`
`registerObsMgr(+H)`	Reregisters OM with handle `H` with its corresponding sensor
`lastObservation(+H, -O)`	Binds the last (most recent) observation stored in the window of OM with handle `H` to `O`
`firstObservation(+H, -O)`	Binds the first (oldest) observation stored in the window of OM with handle `H` to `O`

An OM either implements the **`SensorEventListener`** interface or the **`LocationListener`** interface, depending on if observations are to be collected from a motion, position, or environmental sensor or from a location provider. OMs store new observations when they are received in an chronologically-ordered collection called a *window*. Each window has a capacity that is either size-based or time-based. A window with a size-based capacity only maintains a certain number of observations while one with a time-based capacity only maintains the observations that occur during a particular duration. The duration is the difference in time between the most recently received observation and the oldest observation currently maintained in the window. When an OM receives a new observation and its window is at capacity, the OM adds the new observation to the window but removes old observations until the window is once again at the proper capacity, as illustrated in figure 4.2.

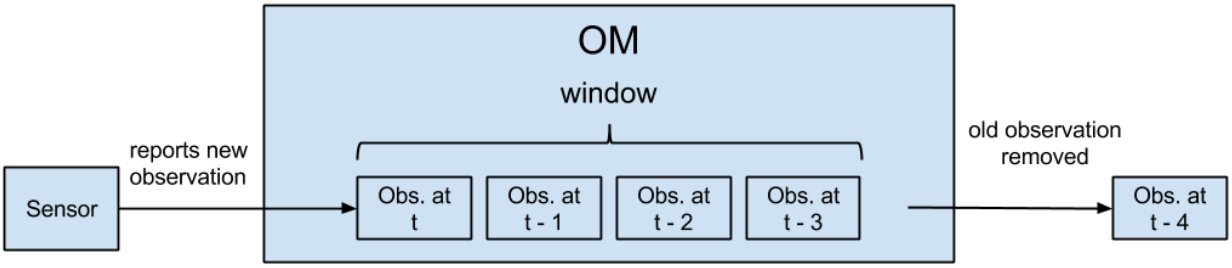

Figure 4.2: Observation manager

An OM can be created using the predicate **createObsMgr(+H, +Type, +Rate)** where **H** is the handle to refer to the OM once it has been created, **Type** is the type of sensor with which to register the OM, and **Rate** is the requested sampling rate of the sensor. For example, the following query will create an OM called **accelOM** for the accelerometer and request that observations are reported every 0.02 seconds (50 Hz):

```
?- createObsMgr(accelOM, accelerometer, 0.02).
```

The requested sampling rate for motion, position, and environmental sensors can be specified using an enumerated constant value or an absolute value. The available constant values in order from slowest to fastest are: **normal** (∼5 Hz), **ui** (∼17 Hz), **game** (∼50 Hz), **fastest** (uncapped; as fast as possible). A specific sampling rate can be requested by passing a positive floating point number representing the delay in seconds in place of the constant[1]. The requested sampling rate for location sensors, however, can only be specified using a positive floating point number representing the delay in seconds[2]. It is important to note that the requested sampling rate is only a suggestion. Sensors may ignore the request and report observations at a rate that is different from the requested rate. When an OM is created using **createObsMgr/3**, SensorLib handles the registeration of the sensor event listener automatically. The created OM will begin immediately storing observations in the sliding window it maintains as they are received.

By default, an OM is created with a window with a size-based capacity of 100 observations. A window's size starts at zero and grows to its set capacity as new sensor observations are reported. The predicate **setCapacity(+H, +CapType, +CapVal)** can be used to modify the capacity type and capacity of the window of an OM at runtime. For example, the following query sets the capacity of the OM **accelOM** created above to a duration of two

[1]The smallest allowed value is 0.000004 because this corresponds to 4 microseconds. Smaller values would conflict with the integer values used by the Android SDK to represent the enumerated constants.

[2]The Android SDK treats location providers differently than sensors and does not provide sampling rate constant values for location providers.

seconds:

```
?- setCapacity(accelOM, time, 2.0).
```

The new capacity type of the window is specified by an enumerated constant. The constant **`size`** indicates the capacity should be size-based and the constant **`time`** indicates the capacity should be time-based. The new capacity of the window is specified using an absolute value. If the new capacity of a window is smaller than the previous capacity, the window is shrunk to the size or duration of the new capacity.

Direct access to arbitrary observations stored in a window that is actively receiving new observations is not permitted by SensorLib because they could be removed from the window at any time. However, SensorLib does allow direct access to the most recent and oldest observations stored in a window. The predicate **`lastObservation(+H, -O)`** retrieves the last, or most recent, observation received by an OM:

```
?- lastObservation(accelOM, O).
```

The observation is stored in a JPR **`ReferenceTerm`** which is bound to the output argument **`O`**. Likewise, the predicate **`firstObservation(+H, -O)`** stores the first, or oldest, observation in the window of an OM in a JPR **`ReferenceTerm`** which is bound to the output argument **`O`**. Predicates for working with observations are outlined in section 4.7

4.5 Window clones

SensorLib provides OMs to acquire, store, and manage the observations reported by on-board sensors in a window of certain capacity. The contents of a window are constantly changing as the observations arrive in a continuous stream. SensorLib does not permit direct access to arbitrary observations in an active window (except for the first and last observations) as they could be removed from the window at any time by the OM. Instead, predicates are

provided for obtaining a clone, or a copy, of the window which can then be iterated over to access each observation chronologically. The predicates for cloning windows and working with window clones can be found in table 4.3.

Table 4.3: Window clone predicates

Predicate	Functionality
`cloneWindow(+H, -W)`	Clones the window of OM with handle `H` and binds the clone to `W` as a `ReferenceTerm`
`cloneWindowNoOverlap(+H, +PrevW, -NewW)`	Clones the window of OM with handle `H` that has no overlapping observations with `PrevW` and binds the clone to `NewW` as a `ReferenceTerm`
`getDuration(+W, -Dur)`	Binds the duration (time of most recently stored observation minus time of oldest stored observation) in seconds of the window clone in the `ReferenceTerm W` to `Dur`
`getSize(+W, -Size)`	Binds the size (number of observations) of the window clone in the `ReferenceTerm W` to `Size`
`getIterator(+W, -Iter)`	Binds an iterator of the window clone in the `ReferenceTerm W` to `Iter` as a `ReferenceTerm`

The predicate **`cloneWindow(+H, -W)`** can be used to obtain a clone of the window of an OM. The window clone is stored in a JPR **`ReferenceTerm`** and bound to the output argument `W`. The window clone is a "shallow" copy of the active window meaning that the object references in the clone point to the same Java observation objects in the active window. As SensorLib only provides read access to observations stored in a window clone, there is no danger of accidental modification of the observations stored in the active window of the OM. Cloning a window consecutively can yield nearly identical clones depending on the rate at which observations are reported. Therefore, SensorLib provides the predicate **`cloneWindowNoOverlap(+H, +PrevW, -NewW)`** that produces a window clone **`NewW`** that contains no observations that are within the window clone **`PrevW`**[3].

An OM that maintains a window with size-based capacity ensures that the window will not contain more observations than specified by the capacity. Likewise, a window with a time-based capacity is ensured to only store observations that are bounded by a specific

[3]Non-overlapping windows are referred to as "tumbling windows" in the literature [64].

duration. The predicate **getDuration(+W, -Dur)** can be used to obtain the duration of a cloned window and the predicate **getSize(+W, -Size)** can be used to obtain its size. **Dur** is bound to a floating point number representing the duration of the window in seconds and **Size** is bound to an integer representing the number of observations contained in the window. The sampling delay of a particular sensor can easily be determined using these predicates together:

```
?- cloneWindow(accelOM, W),
   getDuration(W, Dur),
   getSize(W, Size),
   Delay is Dur / Size.
```

SensorLib provides access to observations stored in window clones through iterators. The predicate **getIterator(+W, -Iter)** binds an iterator of a window clone to **Iter** as a **ReferenceTerm**. The iterator can then be used in conjunction with the **next/2** predicate to access each observation stored in the window clone. For example, the following rules compute the mean of values in a window clone:

```
% computes mean values of observations in window clone W
mean(W, [X, Y, Z]) :-
  getIterator(W, Iter),
  sum(Iter, [0, 0, 0],  [Sx, Sy, Sz]),
  getSize(W, Size),
  X is Sx / Size, Y is Sy / Size, Z is Sz / Size.

% sums values of observations visited by iterator Iter
sum(Iter, [Px, Py, Pz], [Sx, Sy, Sz]) :-
  next(Iter, Obs),
  getValues(Obs, [X, Y, Z]),
  Tx is Px + X, Ty is Py + Y, Tz is Pz + Z,
  sum(Iter, [Tx, Ty, Tz], [Sx, Sy, Sz].

% base case; iterator has no more observations to visit
sum(_, Vals, Vals) :- !.
```

SensorLib does not allow for iterators to the active window to be obtained. If such an iterator were to be provided, the observation it pointed to could be removed from the active window. All the observations between the current iterator observation and the oldest observation

maintained in the active window would remain in memory thus negating the low-memory advantage of the OM.

4.6 Built-in statistical operations

For convenience, SensorLib provides a number of predicates for performing common statistical operations on window clones. The built-in statistical operations are all implemented in Java for efficiency. Table 4.4 lists the built-in statistical operations that operate on the values of observations.

Table 4.4: Built-in statistical operation predicates

Predicate	Functionality
`minValues(+W, [-V1, ..., -Vn])`	Binds minimum values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list
`maxValues(+W, [-V1, ..., -Vn])`	Binds maximum values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list
`medianValues(+W, [-V1, ..., -Vn])`	Binds median values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list
`meanValues(+W, [-V1, ..., -Vn])`	Binds mean of values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list
`varianceValues(+W, [-V1, ..., -Vn])`	Binds variance of values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list
`stddevValues(+W, [-V1, ..., -Vn])`	Binds minimum value all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

In addition to operations on values, SensorLib provides predicates that perform similar operations on the magnitude of observation value vectors (listed in appendix B). These magnitude operations are useful for motion and position sensors because as the orientation of a device changes, the coordinate system of the sensors that use a device-oriented frame of reference rotates accordingly and the values of the observations along the three axes will also change. However, the magnitude of a sensor's observation value vector indicates the quantity of the measurement and has no directions, therefore is orientation-dependent [65].

4.7 Observations

The Android SDK treats motion, position, and environmental sensors differently than the GPS and network location providers. In particular, the underlying Java objects used to represent observations of the two types of sensors are different which means that the data associated with the two types of observations is represented differently. SensorLib however provides a uniform view of sensors and location providers and thus provides a uniform view of their reported observations. Each observation possesses the values of the observation, the time at which the observation was made, the accuracy of the observation, and the sensor that reported the observation. The predicates for retreiving data stored in an observation can be found in table 4.5.

Table 4.5: Observation predicates

Predicate	Functionality
`getValues(+O, [-V1, ..., -Vn])`	Binds each value of the observation in the `ReferenceTerm O` to unbound variables `V1` to `Vn` in the Prolog list
`getTime(+O, -Time)`	Binds the timestamp of the observation in the `ReferenceTerm O` to `Time`
`getAccuracy(+O, -Accy)`	Binds the accuracy of the observation in the `ReferenceTerm O`
`getSensor(+O, -S)`	Binds the type of the sensor that reported the observation in the `ReferenceTerm O` to `S` as a sensor type atom

Different sensors deliver a different number of values with each reported observation. For example, an accelerometer measures the current acceleration of the device along its x, y, and z axes therefore delivers three values with each observation. A light sensor, on the other hand, only measures illuminance so it only reports one value per observation. SensorLib provides uniform access to the values of the observations with the predicate **`getValues(+O, [-V1, ..., -Vn])`**. The length of the output argument list is dependent on the type of sensor that reported the observation:

```
getValues(AccelObs, [X, Y, Z])
getValues(LightObs, [Illum])
getValues(GpsObs, [Lat, Long])
```

The predicate **getTime(+O, -Time)** binds the time in nanoseconds of an observation to **Time**. The times of two different observations can be compared using the standard Prolog comparison operators:

```
getTime(O1, T1),
getTime(O2, T2),
T1 < T2,            % true if O1 occurred before O2
...
```

As the specifications do not indicate what reference point the motion, position, and environment sensors use to indicate time, only observations reported by the same sensor should be compared. However, the time associated with the observations reported by location providers is defined as the number of nanoseconds since the device was last booted[4].

The predicate **getAccuracy(+O, -Accy)** is provided for extracting the accuracy associated with an observation. If the observation was reported by a motion, position, or environmental sensor, **Accy** will be bound to one of **high** (sensor is reporting observations with maximum accuracy), **medium** (sensor is reporting observations with an average level of accuracy), **low** (sensor is reporting observations with low accuracy), or **unreliable** (values of the observations reported by the sensor cannot be trusted) depending on the reported accuracy of the observation. If instead the observation was reported by a location provider, **Accy** will be bound to a floating point number indicating the accuracy in meters of the 68% confidence radius.

The value representing the accuracy of an observation maps directly to a similar constant value defined by the Android SDK. Unfortunately, the Android SDK documentation gives no indication of how to interpret these values. Furthermore, it seems as if this accuracy value is not entirely trustworthy. For instance, the accelerometer of the device used in testing (an HTC One Remix) always reported an "unreliable" accuracy though the accelerometer properly reported the force of gravity while the device remained at rest. This suggests that

[4]The reason behind SensorLib's version require of Android 4.2 (API level 17) is due to the use of a method for obtaining the time of a reported **Location** that is not compatible with earlier versions.

it is not the case that the accuracy of the accelerometer readings was "unreliable" but rather that the reported accuracy was incorrect. The accuracy of observations reported by location providers do not suffer from these problems.

4.8 Summary

SensorLib is a Prolog library that facilitates the aquisition of observations reported by sensors on Android devices. Observation managers (OMs) are registered with sensors and automatically store observations when they are reported. The sliding window technique implemented by OMs emphasizes that recent data is more relevant than old data [19] while using only enough memory required to store the desired observations. The observations are accessible by iterating over a clone of the active window managed by an OM, though SensorLib also provides built-in operations for computing statistical summaries of the clones.

Chapter 5

Activity recognition using OwlLib and SensorLib

5.1 Introduction

This thesis has produced two Prolog libraries capable of accessing two external sources of information. OwlLib provides access to OWL ontologies while SensorLib provides access to sensor observations reported by on-board sensors of Android-powered mobile devices. Both libraries were developed independently and they perform seperate functions. Nevertheless, they can be used together and this chapter presents demonstrations of the collaborative use of OwlLib and SensorLib along with Prolog rules to perform useful tasks related to physical activity recognition. The examples presented in this chapter are not meant to constitute a full and complete Android application but rather to demonstrate the usefulness of the products of this thesis.

The demonstrations in this chapter discuss how SensorLib is used to continuously aquire sensor observations reported by on-board sensors over a small time window. The raw sensor values associated with the observations are processed into more useful information using

the built-in statistical operations provided by SensorLib. The processed observation data is passed to Prolog rules that determine the user's current physical activity, such as standing, walking, jogging, or ascending and descending stairs. The Prolog rules are derived from a decision tree that is created offline using a publically available data set [66, 67]. OwlLib is used to create a "semantic" log of the occurence of activities by extending two popular ontologies, one for describing sensors and their observations and the other for describing time based on intervals.

This chapter is organized as follows. Section 5.2 describes the data set that is used to derive the training instances used to create a decision tree for performing activity recognition. Details regarding the creation of the decision tree and its conversion into Prolog rules are outlined in section 5.3. The use of SensorLib to aquire sensor observations and compute statistical summaries regarding them is discussed in section 5.4. The ontology for describing sensors and their observations and the ontology for describing time based on intervals are introduced in section 5.5, as well as the the ontology that extends these ontologies and the use of the ontology to qualitatively relate detected activities and store them in a "semantic" log.

5.2 The data set and training instances

Shoaib et al. [67] recently investigated to what extent sensors that are commonly available on mobile devices are capable of detecting various physical activities. In particular, they were interested in comparing how the accelerometer and the gyroscope performed when used in isolation and when used together. The seven activities that were investigated were standing, walking, ascending stairs, descending stairs, jogging, bicycling, and sitting. Several Android-powered devices were positioned at various common carrying positions and data was collected at a rate of 50 Hz from the observations reported by the accelerometers and gyroscopes. The

raw values of the sensor observations were made publically available online [66]. For this demonstration, the data set was used to derive training instances to create a decision tree that will be converted into Prolog rules for activity recognition. The activities considered for this demonstration are standing, walking, jogging, and ascending and descending stairs when measured by an Android device carried in the front right pocket.

The raw data values of the sensor observations were partitioned into two-second time windows. This choice of window duration is supported by the results of similar work [68] that compared window durations of one, two, five, and ten seconds for detecting cyclic activities and concluded that windows with two second durations yielded the best classification performance. A training instance was created based on statistical summaries of each window yielding 16 features, namely the means and standard deviations of the values of the x, y, and z axes and the magnitudes of the value vectors for the accelerometer and the gyroscope. When partitioned in this manner, there were 810 training instances per activity, totalling 4050 training instances for all five activities.

5.3 Detecting activities using Prolog rules

In order to derive Prolog rules capable of performing activity recognition, a decision tree was created in the WEKA machine learning suite [69] using the training instances described in section 5.2. Based on the learned tree, a Prolog rule can be created from each path from the root node to a leaf node. The default settings of the J48 decision tree algorithm included with WEKA produced a decision tree with an overall accuracy of 97% for classifying all five activities. However, it consisted of 61 leaves. Since each leaf corresponds to a Prolog rule, the translation of this tree into a rule set yields a set of Prolog rules that is arguably too specific.

Experiments were conducted in order to minimize the number of leaves in the learned

tree while still maintaining acceptable overall accuracy. The default settings of J48 include the specification of assigning a minimum of two training instances per leaf. To produce trees with less leaves, various values of the minimum-instances-per-leaf parameter were used ranging from 100 to 800. The graph in figure 5.1 illustrates how the overall accuracy and number of leaves are affected by modifying the minimum number of instances assigned to each leaf.

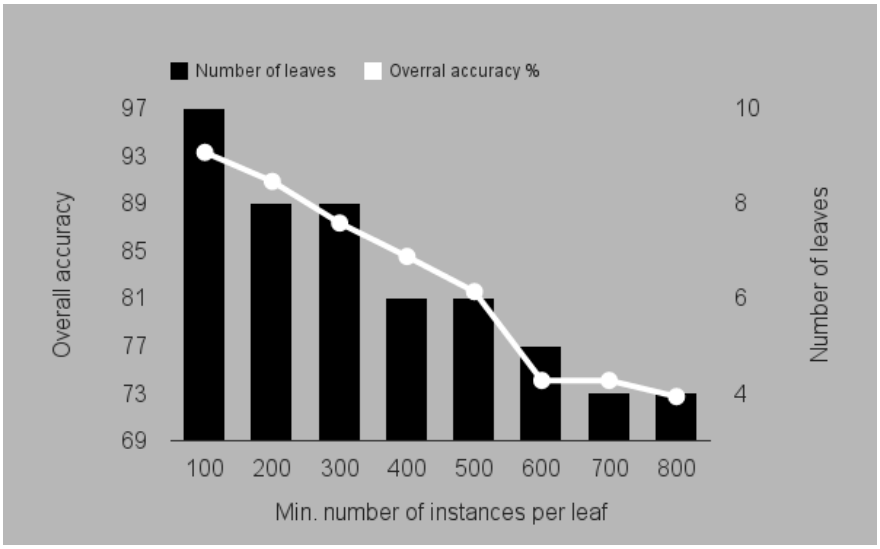

Figure 5.1: Decision tree configurations. This graph illustrates the overall accuracy and number of leaves vs. minimum number of instances per leaf. Note that the increase of assigning 400 instances per leaf as opposed to 300 instance per leaf decreased the number of required rules from eight to six with only about a 3% drop in accuracy. When 500 instances per leaf were used, there was another 3% drop in accuracy without the corresponding decrease in the number of rules required to represent the tree.

Ultimately, the tree with a minimum of 400 instances per leaf yielded the greatest percentage of accuracy per rule and was chosen as the candidate for translation to Prolog rules. A textual representation of the learned decision tree is listed in figure 5.2.

```
sdAmag <= 1.307411: standing
sdAmag > 1.307411
|   sdAy <= 5.868613
|   |   meanAz <= -1.473581
|   |   |   meanGmag <= 1.829171: downstairs
|   |   |   meanGmag > 1.829171: upstairs
|   |   meanAz > -1.473581
|   |   |   sdAmag <= 3.943164: upstairs
|   |   |   sdAmag > 3.943164: walking
|   sdAy > 5.868613: jogging
```

Figure 5.2: Decision tree for physical activity recognition

Note that of the 16 features included with each training instance, only four were used to create the tree: the standard deviation of the accelerometer vector magnitude (**sdAmag**), the standard deviation of the y-axis acceleromter value (**sdAy**), the mean of the z-axis accelerometer value (**meanAz**), and the mean of the gyroscope vector magnitude (**meanGmag**). Based on the features that were used, the accelerometer contributed more to activity recognition than did the gyroscope, but the gyroscope did assist in differentiating between ascending and descending stairs which aligns with the results presented in [67].

```
detect(SdAmag, _, _, _, standing) :-
  SdAmag =< 1.307411.

detect(SdAmag, SdAy, MeanAz, MeanGMag, downstairs) :-
  SdAmag > 1.307411, SdAy =< 5.868613, MeanAz =< -1.473581,  MeanGmag =< 1.829171.

detect(SdAmag, SdAy, MeanAz, MeanGMag, upstairs) :-
  SdAmag > 1.307411, SdAy =< 5.868613, MeanAz =< -1.473581, MeanGmag > 1.829171.

detect(SdAmag, SdAy, MeanAz, MeanGMag, upstairs) :-
  SdAmag > 1.307411, SdAy =< 5.868613, MeanAz > -1.473581, SdAmag =< 3.943164.

detect(SdAmag, SdAy, MeanAz, MeanGMag, walking) :-
  SdAmag > 1.307411, SdAy =< 5.868613, MeanAz > -1.473581, SdAmag > 3.943164.

detect(SdAmag, SdAy, _, _, jogging) :-
  SdAmag > 1.307411, SdAy > 5.868613.
```

Figure 5.3: Prolog rules for physical activity recognition

Since the learned decision tree contains six leaves, it can be converted to Prolog using the six rules as shown in figure 5.3. Each path from the root of the decision tree to a leaf corresponds to a Prolog rule which can be read naturally as numerical comparisons of the input parameters. For example, the second rule in figure 5.3 reads: "If the standard deviation of the magnitudes of the accelerometer (**SdAmag**) is greater than 1.307411 and the standard deviation of the y-axis values of the accelerometer (**SdAy**) is less than or equal to 5.868613 and the mean of the z-axis values of the accelerometer (**MeanAz**) is less than or equal to -1.473581 and the mean of the magnitudes of the gyroscope (**MeanGmag**) is less than or equal to 1.829171, then the current activity is descending stairs (**downstairs**)".

5.4 Collecting sensor observations using SensorLib

The Prolog rules in figure 5.3 are capable of classifying the most recent physical activity performed by an individual using the accelerometer and gyroscope of a mobile device located

in the right front pants pocket. However, the rules require certain statistical summaries regarding the last two seconds of observations reported by the sensors to perform the classification. SensorLib can be used to aquire the necessary observations and compute the required statistical summaries.

The following two queries first verify the presence of an accelerometer and gyroscope, then create an observation manager for each sensor and requesting that each sensor report a value every 0.02 seconds (equivalent to 50 Hz), and lastly set the window capacity to a duraton of two seconds.

```
?- hasSensor(accelerometer),
   createObsMgr(accelOM, accelerometer, 0.02),
   setCapacity(accelOM, time, 2.0).
?- hasSensor(gyroscope),
   createObsMgr(gyroOM, gyroscope, 0.02),
   setCapacity(gyroOM, time, 2.0).
```

These settings correspond to the training instances used to create the decision tree. If a device lacks a gyroscope, the rules from figure 5.3 requiring statistics regarding gyroscope observations can be modified or omitted as desired (then the second query need not be made at all). With the observation managers created and properly configured, the last two seconds of observations reported by the sensors will be received, stored, and managed in memory.

SensorLib provides the required operations for computing the required statistical summaries from the stored observation data "out-of-the-box". The query below clones the windows managed by the observation managers, computes the statistical summary values, and passes the values to the activity recognition rules from figure 5.3. Ultimately, the variable **`Activity`** is bound to the classified activity. If the device lacks a gyroscope, the predicates involving the gyroscope could be omitted from the query. The query can be called repeatedly every two seconds to provide continuous activity recognition.

```
?- cloneWindow(accelOM, Wa),
   cloneWindow(gyroOM, Wg),
   stddevMagnitude(Wa, SdAmag),
   stddevValues(Wa, [_, SdAy, _]),
   meanValues(Wa, [_, _, MeanAz]),
   meanMagnitude(Wg, MeanGmag),
   detect(SdAmag, SdAy, MeanAz, MeanGmag, Activity).
```

5.5 Logging activities using OwlLib

While SensorLib along with the activity recognition rules can be used to detect a user's current activity at a particular time or continuously monitor a user's activity, it would be useful to maintain a record or log of detected activities. Such a log could be analyzed to infer further information that is not immediately apparent when processing the two-second windows at the time they are obtained. OwlLib can be used to create a "semantic" log in which records of a user's activity are not simply recorded but are also qualitatively related to one another. The method outlined here will be to assert axioms that express details regarding the windows of observations into an ontology that extends two popular ontologies, one for describing sensors and their observations and the other for describing time based on intervals.

The Semantic Sensor Network (SSN) ontology [33] developed by the W3C Semantic Sensor Network Incubator Group is an OWL ontology for describing sensors and their observations. It defines a Sensor generically as anything that implements some sensing and thus observes some Property of a FeatureOfInterest. While the accelerometer and gyroscope are obviously sensors, the combination of the activity recognition rules and SensorLib can itself be defined as a sensor that observes the current activity of the user based on statistical summaries of a two-second window of observations.

The OWL-Time ontology [34, 70] is a working draft of an OWL ontology published by the W3C for describing instants and intervals of time. It defines a ProperInterval as a Tempo-

ralEntity with extent (therefore, not an single point in time). It also defines object properties for describing the relationships of ProperIntervals to one another based on Allen's interval algebra [35]. For instance, intervalAfter relates one ProperInterval as occurring completely after another with no overlap and intervalDuring relates one ProperInterval as occurring completely during another with no overlap.

The two-second windows of observations can be classified as ProperIntervals and each can be connected to another by a before-after relationship forming a linked list of intervals. At a higher level, multiple consecutive atomic windows indicative of the same activity can be viewed collectively as a single larger compound window that is also a ProperInterval. Each of the individual atomic windows can be connected to the compound window by a during-contains relationship. Also, the first atomic window can be connected to the compound window by a starts-startedBy relationship while the last atomic window can be connected to it by a finishes-finishedBy relationship. Furthermore, compound windows can also be connected to one another by a before-after relationship, forming another linked list of intervals containing intervals.

Based on the above observations, the ontology illustrated in figure 5.4 was defined using the SSN ontology and OWL-Time. Standing, walking, jogging, etc. are instances of Activity that is a subclass of Property of the SSN ontology and thereby are observable by the activity recognition rules which are classified together as a Sensor as discussed above. Every Window (atomic or compound) indicates one of the activities. Also, since each Window is also a ProperInterval, they can be related to each other by the listed qualitative properties.

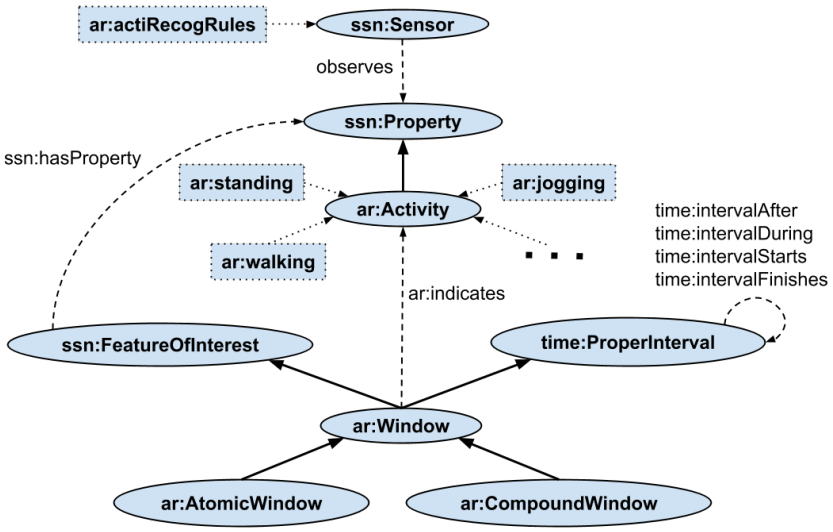

Figure 5.4: Ontology for logging activity recognition windows

A graphical representation of a log that can be created using the ontology is illustrated in figure 5.5. As each two-second window is collected and processed by SensorLib and classified by the activity recognition rules, it can be asserted into the ontology as being indicative of the classified activity and related as occurring after the previous atomic window. As long as the rules detect the same activity repeatedly, each atomic window can be related as occurring during the current compound window. When the activities change, the current compound window is finished by the previous atomic window and a new compound window indicative of the new activity is started by the new atomic window. The log in the figure represents an interval of standing followed by an interval of walking. The compound window **cw459** indicates standing because the atomic windows **aw1381** and **aw1382**, which both occur during it, indicate standing. Similarly, the compound window **cw460** indicates walking because the atomic windows that comprise it indicate walking.

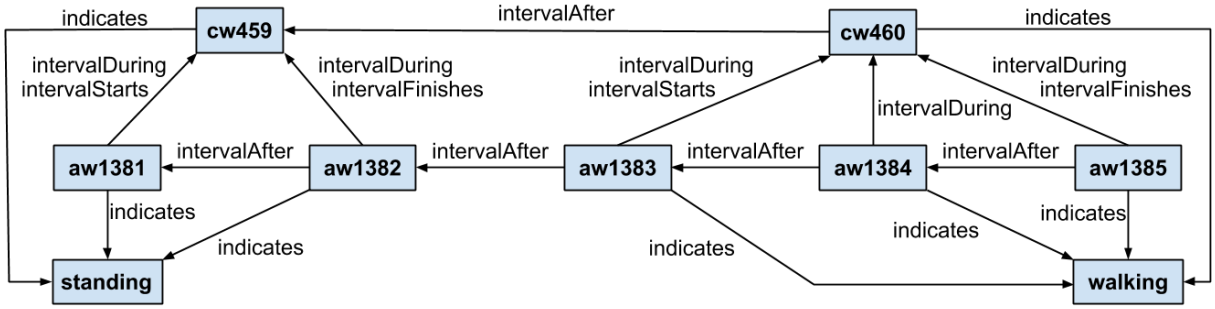

Figure 5.5: Semantic log of activities

5.6 Conclusion

This chapter demonstrated how SensorLib and OwlLib could be used along with Prolog rules to detect the current activity being performed by an individual with a mobile device placed in a pants pocket. SensorLib was used to aquire and process sensor observations, Prolog rules were used to detect the user's activity based on the processed observations, and OwlLib was used to create a "semantic" log of the activities. Applications of physical activity recognition of this kind include fitness tracking, health monitoring, and self-managing systems, among others [71]. Though these demonstrations do not encompass the extent that OwlLib and SensorLib can be used in conjunction with Prolog rules nor they intended to be robust enough to be used in practice, they do demonstrate how easy it is to use both together effectively.

Chapter 6

Conclusions and Future Work

This thesis has produced two Prolog libraries, OwlLib and SensorLib. Each library is designed for integrating a different form of external information with logic programs. As both libraries are designed for an Android-compatible Prolog environment, they open many doors for creating knowledge-based systems that can take advantage of the ubiquity of mobile devices. OwlLib provides access to computable inferences of OWL ontologies through the use of external reasoners thus enabling the reuse of knowledge expressed in ontologies from within Prolog programs. SensorLib provides access to observations reported by the on-board sensors of mobile devices thus enabling the creation of context-aware applications.

It has been shown how OwlLib and SensorLib can be used in conjunction with Prolog rules to perform useful tasks. Demonstrative examples of how the two libraries could be used along with Prolog rules to detect the current activity being performed by an individual with a mobile device have been provided. SensorLib was used to aquire and process sensor observations, Prolog rules were used to detect the user's activity based on the processed observations, and OwlLib was used to create a "semantic" log of the activities. Possible future applications of using the libraries to perform tasks of this kind related to physical activity recognition include fitness tracking, health monitoring, and self-managing systems,

among others [71].

Possible future works related to OwlLib include a complete abstraction from the dependencies on the OWL API and possibly even OWL itself. While it is possible to replace the OWL API with another framework for working with OWL ontologies and reasoners, such a venture would require extensive rewriting. It would be beneficial to decouple OwlLib from the OWL API in such a way that the use of the framework is completely transparent. This venture would also involve another solution for uniformly accessing external reasoners that is not dependent on the interface supplied by the OWL API. Another interesting generalization would be the modification and rebranding of OwlLib to work with DL ontologies at a higher level that is free of specifics of OWL. Alternatively, this generalization could be realized by another library for JPR named something along the lines of *DlLib* or something similar. Yet another abstraction would be a library for JPR that provided a uniform way to access other external data sources, much like HEX-programs [42].

Possible future works related to SensorLib involve the inclusion other apparatus common on Android devices such as cameras and microphones. Cameras are able to capture still images and video. The processing of these two types of media fall under the fields of image processing and computer vision. The observations reported by cameras can be used Microphones are able to capture streams of auditory signals. Such observations are useful in fields such as speech analysis. The introduction of cameras and microphones might require an alternative mechanism for accessing their reported observations as the data values may be highly dimensional. A picture captured by a camera for instance is representable by an image consisting of hundreds of pixels. The current mechanism provided by SensorLib for obtaining the values of an observation almost certainly would not suffice.

Appendices

Appendix A

Listing of OwlLib predicates

A.1 Ontology management predicates

`createOntology(+O)`
Creates an empty ontology with the handle `O`

`loadOntology(+Path, +O)`
Loads an ontology with the handle `O` from a file on the path `Path` of the file system

---

`useOntology(+O)`
Sets the ontology with the handle `O` to be the active ontology

---

`ontologyLoaded(+O)`
Determines if an ontology with the handle `O` has been loaded

---

`activeOntology(-O)`
Binds `O` to the handle of the active ontology

`loadedOntologies(-Os)`
Binds `Os` to a Prolog list containing the handles of all loaded ontologies

---

`saveOntology(+Path, +O)`
Saves the ontology with the handle `O` to a file with the path `Path` on the file system

---

`unloadOntology(+O)`
Unloads the ontology with the handle `O`

A.2 Reasoner management predicates

`createReasoner(+R, +Type, +O)`
Creates an reasoner with the handle `R` of the type `Type` for the ontology with the handle `O`

`useReasoner(+R, +O)`
Sets the reasoner with the handle `R` to be the active reasoner for the ontology with the handle `O`

`activeReasoner(+O, -R)`
Binds `R` with the handle of the active reasoner of the ontolog with the handle `O`

`loadedReasoners(+O, -Rs)`
Binds `Rs` to a Prolog list containing the handles of all the loaded reasoners for the ontology with the handle `O`

`removeReasoner(+R, +O)`
Removes the reasoner with the handle `R` from the ontology with the handle `O`

A.3 Entity terms

`namedIndividual(+IRI)`
Indicates the entity represented by `IRI` is a named individual

---

`class(IRI)`
Indicates the entity represented by `IRI` is an atomic class

`objectProperty(IRI)`
Indicates the entity represented by `IRI` is an object property

---

A.4 Expression terms

A.4.1 Class expression terms

`objectIntersectionOf(+CEs)`

Represents the set of individuals that are instances of every class expression in the Prolog list `CEs`

`objectUnionOf(+CEs))`
Represents the set of individuals that are instances of at least one of the class expressions in the Prolog list `CEs`

`objectComplementOf(+CE)`
Represents the set of individuals that are not instances of the class expression `CE`

`objectOneOf(+Is)`
Represents the set of individuals in the Prolog list `Is`

`objectSomeValuesFrom(+OPE, +CE)`
Represents the set of individuals that are connected by `OPE` to an individual that is an instance of `CE`

`objectAllValuesFrom(+OPE, +CE)`
Represents the set of individuals that are connected by `OPE` only to individuals that are instances of `CE`

`objectHasValue(+OPE, +I)`
Represents the set of individuals that are connected by `OPE` to `I`

`objectHasSelf(+OPE)`
Represents the set of individuals that are connected by `OPE` to themselves

---

`objectMinCardinality(+N, +OPE, +CE)`
Represents the set of individuals that are connected by `OPE` to at least `N` different individuals that are instances of `CE`

`objectMaxCardinality(+N, +OPE, +CE)`
Represents the set of individuals that are connected by `OPE` to at most `N` different individuals that are instances of `CE`

`objectExactCardinality(+N, +OPE, +CE)`
Represents the set of individuals that are connected by `OPE` to exactly `N` different individuals that are instances of `CE`

`dataSomeValuesFrom(+DPEs, +DR)`
Represents the set of individuals that are connected by data property expressions in the Prolog list `DPEs` to literals that are within `DR`

---

dataAllValuesFrom(+DPEs, DR)
Represents the set of individuals that are connected by data property expressions in the Prolog list `DPEs` only to literals that are within `DR`

---

dataHasValue(+DPE, +Lit)
Represents the set of individuals that are connected by `DPE` to `Lit`

---

dataMinCardinality(+N, +DPE, +DR)
Represents those individuals that are connected by `DPE` to at least `N` different literals in `DR`

---

dataMaxCardinality(+N, +DPE, +DR)
Represents those individuals that are connected by `DPE` to at most `N` different literals in `DR`

---

dataExactCardinality(+N, +DPE, +DR)
Represents those individuals that are connected by `DPE` to exactly `N` different literals in `DR`

A.4.2 Data range terms

dataIntersectionOf(+DRs)
Contains all tuples of literals that are contained in each data range in the Prolog list `DRs`

---

dataUnionOf(+DRs)
Contains all tuples of literals that are connected in at least one data range in the Prolog list `DRs`

---

dataComplementOf(+DR)
Contains all tuples of literals that are not contained in data range `DR`

---

dataOneOf(+Lits)
Contains exactly the literals in the Prolog list `Lits`

A.4.3 Object property expression terms

inverseObjectProperty(OP)
Represents the opposite relationship of the relationship represented by the object property `OP`

A.5 Axiom predicates

A.5.1 Class expression axiom predicates

`subClassOf(?CE1, ?CE2)`
States that `CE1` is subsumed by `CE2`

`equivalentClasses(+CEs)`
States that class expressions in the Prolog list `CEs` are equivalent

---

`disjointClasses(+CEs)`
States that class expressions in the Prolog list `CEs` are pairwise disjoint

---

`disjointUnion(+C, +CEs)`
States that the class `C` is a disjoint union of the class expressions in the Prolog list `CEs`

A.5.2 Object property expression axiom predicates

`subObjectPropertyOf(?OPE1, ?OPE2)`
States that `OPE1` is a subproperty of `OPE2`

`equivalentObjectProperties(+OPEs)`
States that object property expressions in the Prolog list `OPEs` are equivalent

---

`disjointObjectProperties(+OPEs)`
States that object property expressions in the Prolog list `OPEs` are pair-wise disjoint

---

`inverseObjectProperties(?OPE1, ?OPE2)`
States that `OPE1` and `OPE2` are inverses

`objectPropertyDomain(+OPE, ?CE)`
States that the domain of `OPE` is the class expression `CE`

`objectPropertyRange(+OPE, ?CE)`
States that the range of `OPE` is the class expression `CE`

`functionalObjectProperty(+OPE)`
States that `OPE` is functional

`inverseFunctionalObjectProperty(+OPE)`

States that `OPE` is inverse-functional

`reflexiveObjectProperty(+OPE)`
States that `OPE` is reflexive

`irreflexiveObjectProperty(+OPE)`
States that `OPE` is irreflexive

`symmetricObjectProperty(+OPE)`
States that `OPE` is symmetric

`asymmetricObjectProperty(+OPE)`
States that `OPE` is asymmetric

`transitiveObjectProperty(+OPE)`
States that `OPE` is transitive

A.5.3 Data property expression axiom predicates

`subDataPropertyOf(?DPE1, ?DPE2)`
States that `DPE1` is a subproperty of `DPE2`

`equivalentDataProperties(+DPEs)`
States that data property expressions in Prolog list `DPEs` are equivalent

---

`disjointDataProperties(+DPEs)`
States that data property expressions in Prolog list `DPEs` are pair-wise disjoint

---

`dataPropertyDomain(+DPE, ?CE)`
States that the domain of `DPE` is the class expression `CE`

`dataPropertyRange(+DPE, +DR)`
States that the range of `DPE` is the data range `DR`

`functionalDataProperty(+DPE)`
States that `DPE` is functional

A.5.4 Assertional expression axiom predicates

`sameIndividual(+Is)`

States that individuals in Prolog list `Is` are all equal to one another

---

`differentIndividuals(+Is)`
States that individuals in Prolog list `Is` are all different from one another

---

`classAssertion(?CE, ?I)`
States that `I` is an instance of class expression `CE`

`objectPropertyAssertion(+OPE, +I1, ?I2)`
States that `I1` is connected by `OPE` to `I2`

`negativeObjectPropertyAssertion(+OPE, +I1, +I2)`
States that `I1` is not connected by `OPE` to `I2`

`dataPropertyAssertion(+DPE, +I1, ?Lit)`
States that `I1` is connected by `DPE` to `Lit`

`negativeDataPropertyAssertion(+DPE, +I1, +Lit)`
States that `I1` is connected by `OPE` to `Lit`

A.6 Query predicates

A.6.1 Query predicates for class expression axioms

`getSubClasses(+CE, -Iter)`
Gets set of named subclasses of class expression `CE` and binds iterator of set to `Iter`

`getSuperClasses(+CE, -Iter)`
Gets set of named superclasses of class expression `CE` and binds iterator of set to `Iter`

`getEquivalentClasses(+CE, -Iter)`
Gets set of named equivalent classes of class expression `CE` and binds iterator of set to `Iter`

`getDisjointClasses(+CE, -Iter)`
Gets set of named disjoint classes of class expression `CE` and binds iterator of set to `Iter`

A.6.2 Query predicates for object property expression axioms

`getSubObjectProperties(+OPE, -Iter)`
Gets set of subproperties of `OPE` and binds iterator of set to `Iter`

`getSuperObjectProperties(+OPE, -Iter)`
Gets set of superproperties of `OPE` and binds iterator of set to `Iter`

`getEquivalentObjectProperties(+OPE, -Iter)`
Gets set of equivalent properties of `OPE` and binds iterator of set to `Iter`

`getDisjointObjectProperties(+OPE, -Iter)`
Gets set of disjoint properties of `OPE` and binds iterator of set to `Iter`

`getInverseObjectProperties(+OPE, -Iter)`
Gets set of inverse properties of `OPE` and binds iterator of set to `Iter`

`getObjectPropertyDomains(+OPE, -Iter)`
Gets set of named classes that are domains of `OPE`

`getObjectPropertyRanges(+OPE, -Iter)`
Gets set of named classes that are ranges of `OPE`

A.6.3 Query predicates for data property expression axioms

`getSubDataProperties(+DPE, -Iter)`
Gets set of subproperties of `DPE` and binds iterator of set to `Iter`

`getSuperDataProperties(+DPE, -Iter)`
Gets set of superproperties of `DPE` and binds iterator of set to `Iter`

`getEquivalentDataProperties(+DPE, -Iter)`
Gets set of equivalent properties of `DPE` and binds iterator of set to `Iter`

`getDisjointDataProperties(+DPE, -Iter)`
Gets set of disjoint properties of `DPE` and binds iterator of set to `Iter`

`getDataPropertyDomains(+DPE, -Iter)`
Gets set of named classes that are domains of `DPE`

A.6.4 Query predicates for assertional axioms

`getSameIndividuals(+I, -Iter)`
Gets set of individuals that are the same as `I`

`getDifferentIndividuals(+I, -Iter)`
Gets set of individuals that are different than `I`

`getTypes(+I, -Iter)`
Gets set of named classes of which `I` belongs and binds iterator of set to `Iter`

`getInstances(+CE, -Iter)`
Gets set of individuals that are instances of `CE` and binds iterator of set to `Iter`

`getObjectPropertyValues(+I, +OPE, -Iter)`
Gets set of individuals that are connected to `I` by `OPE` and binds iterator of set to `Iter`

A.7 Ontology manipulation predicates

`assertAxiom(+A, +O)`
Adds the axiom represented by `A` into the ontology with handle `O`

`retractAxiom(+A, +O)`
Removes the axiom represented by `A` from the ontology with handle `O`

`containsAxiom(+O, +A)`
Determines if the ontology with handle `O` contains the axiom represented by `A`

`entailsAxiom(+O, +A)`
Determines if the ontology with handle `O` entails the axiom represented by `A`

`isConsistent(+O)`
Determines if the ontology with handle `O` is consistent

Appendix B

Listing of SensorLib predicates

B.1 Sensor type atoms

`accelerometer`
Represents an accelerometer

`gravitySensor`
Represents a gravity sensor

`gyroscope`
Represents a gyroscope

`linearAccelerationSensor`
Represents a linear acceleration sensor

`rotationVectorSensor`
Represents a rotation vector sensor

`gameRotationVectorSensor`
Represents a game rotation vector

`geomagneticRotationVectorSensor`
Represents a geomagnetic rotation vector

`magneticFieldSensor`
Represents a magnetic field sensor

proximitySensor
Represents a proximity sensor

ambientTemperatureSensor
Represents an ambient temperature sensor

lightSensor
Represents a light sensor

pressureSensor
Represents a pressure sensor

relativeHumiditySensor
Represents a relative humidity sensor

gps
Represents a GPS location provider

networkLocationSensor
Represents a network location provider

B.2 Available sensor predicates

getSensorTypes(-Types)
Binds a Prolog list containing sensor type atoms of available sensors to `Types`

hasSensor(+Type)
Evaluates to true if a sensor of the type represented by `Type` is available on the device and false otherwise

B.3 Observation manager predicates

createObsMgr(+H, +Type, +Rate)
Creates an OM with handle `H` and registers it with sensor of type `Type` with requested sampling rate `Rate`

setCapacity(+H, +CapType, +CapVal)

Sets the capacity type of OM with handle `H` to `CapType` (either `size` or `time`); capacity set to value `CapVal` (either floating point number representing time or integer representing size)

`destroyObsMgr(+H)`
Unregisters OM with handle `H` and discards it completely

`unregisterObsMgr(+H)`
Unregisters OM with handle `H`

`registerObsMgr(+H)`
Reregisters OM with handle `H` with its corresponding sensor

---

`lastObservation(+H, -O)`
Binds the last (most recent) observation stored in the window of OM with handle `H` to `O`

`firstObservation(+H, -O)`
Binds the first (oldest) observation stored in the window of OM with handle `H` to `O`

B.4 Window clone predicates

`cloneWindow(+H, -W)`
Clones the window of OM with handle `H` and binds the clone to `W` as a `ReferenceTerm`

`cloneWindowNoOverlap(+H, +PrevW, -NewW)`
Clones the window of OM with handle `H` that has no overlapping observations with `PrevW` and binds the clone to `NewW` as a `ReferenceTerm`

`getDuration(+W, -Dur)`
Binds the duration (time of most recently stored observation minus time of oldest stored observation) in seconds of the window clone in the `ReferenceTerm` `W` to `Dur`

`getSize(+W, -Size)`
Binds the size (number of observations) of the window clone in the `ReferenceTerm` `W` to `Size`

`getIterator(+W, -Iter)`
Binds an iterator of the window clone in the `ReferenceTerm` `W` to `Iter` as a `ReferenceTerm`

B.5 Statistical operation predicates

`minValues(+W, [-V1, ..., -Vn])`
Binds minimum values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`maxValues(+W, [-V1, ..., -Vn])`
Binds maximum values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`medianValues(+W, [-V1, ..., -Vn])`
Binds median values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`meanValues(+W, [-V1, ..., -Vn])`
Binds mean of values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`varianceValues(+W, [-V1, ..., -Vn])`
Binds variance of values of all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`stddevValues(+W, [-V1, ..., -Vn])`
Binds minimum value all observations in window clone `W` to unbound variables `V1` to `Vn` in the Prolog list

---

`minMagnitude(+W, -Mag)`
Binds minimum magnitude of all observations in window clone `W` to `M`

`maxMagnitude(+W, -Mag)`
Binds maximum magnitude of all observations in window clone `W` to `M`

`medianMagnitude(+W, -Mag)`
Binds median values of all observations in window clone `W` to `M`

`meanMagnitude(+W, -Mag)`
Binds mean of values of all observations in window clone `W` to `M`

`varianceMagnitude(+W, -Mag)`
Binds variance of values of all observations in window clone `W` to `M`

`stddevMagnitude(+W, -Mag)`

Binds minimum value all observations in window clone `W` to `M`

B.6 Observation predicates

`getValues(+O, [-V1, ..., -Vn])`
Binds each value of the observation in the `ReferenceTerm` `O` to unbound variables `V1` to `Vn` in the Prolog list

---

`getTime(+O, -Time)`
Binds the timestamp of the observation in the `ReferenceTerm` `O` to `Time`

`getAccuracy(+O, -Accy)`
Binds the accuracy of the observation in the `ReferenceTerm` `O`

`getSensor(+O, -S)`
Binds the type of the sensor that reported the observation in the `ReferenceTerm` `O` to `S` as a sensor type atom

Bibliography

[1] International Data Corporation (IDC), "Smartphone OS Market Share, Q4 2014." `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`. Accessed: April 2015.

[2] W3C OWL Working Group, "OWL 2 Web Ontology Language Document Overview (Second Edition)." `http://www.w3.org/TR/owl-overview/`. Accessed: April 2015.

[3] C. Baral and M. Gelfond, "Logic programming and knowledge representation," *The Journal of Logic Programming*, vol. 19âĂŞ20, Supplement 1, no. 0, pp. 73 – 148, 1994. Special Issue: Ten Years of Logic Programming.

[4] F. Baader and W. Nutt, "Basic Description Logics," in *The Description Logic Handbook: Theory, Implementation, and Applications* (F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, eds.), New York, NY, USA: Cambridge University Press, 2003.

[5] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf, "Al-log: Integrating datalog and description logics," *Journal of Intelligent Information Systems*, vol. 10, no. 3, pp. 227–252, 1998.

[6] A. Y. Levy and M.-C. Rousset, "Combining horn rules and description logics in CARIN," *Artificial Intelligence*, vol. 104, no. 1âĂŞ2, pp. 165 – 209, 1998.

[7] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: Combining logic programs with description logic," in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, (New York, NY, USA), pp. 48–57, ACM, 2003.

[8] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," w3c member submission, World Wide Web Consortium, 2004.

[9] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "Swrl: A semantic web rule language combining owl and ruleml," w3c member submission, World Wide Web Consortium, 2004.

[10] B. Motik, U. Sattler, and R. Studer, "Query answering for owl-dl with rules," in *The Semantic Web âĂŞ ISWC 2004* (S. McIlraith, D. Plexousakis, and F. van Harmelen, eds.), vol. 3298 of *Lecture Notes in Computer Science*, pp. 549–563, Springer Berlin Heidelberg, 2004.

[11] T. Eiter, G. Ianni, A. Polleres, R. Schindlauer, and H. Tompits, "Reasoning with rules and ontologies," in *Reasoning Web*, vol. 4126 of *Lecture Notes in Computer Science*, pp. 93–127, Springer Berlin Heidelberg, 2006.

[12] R. Rosati, "Integrating ontologies and rules: Semantic and computational issues," in *Reasoning Web* (P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler, eds.), vol. 4126 of *Lecture Notes in Computer Science*, pp. 128–151, Springer Berlin Heidelberg, 2006.

[13] B. Motik and R. Rosati, "Reconciling description logics and rules," *J. ACM*, vol. 57, pp. 30:1–30:62, June 2008.

[14] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits, "Combining answer set programming with description logics for the semantic web," *Artificial Intelligence*, vol. 172, no. 12âĂŞ13, pp. 1495 – 1539, 2008.

[15] T. Eiter, G. Ianni, T. Lukasiewicz, and R. Schindlauer, "Well-founded semantics for description logic programs in the semantic web," *ACM Trans. Comput. Logic*, vol. 12, pp. 11:1–11:41, January 2011.

[16] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, pp. 34–43, May 2001.

[17] A. K. Dey, "Understanding and using context," *Personal Ubiquitous Comput.*, vol. 5, pp. 4–7, Jan. 2001.

[18] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *Int. J. Ad Hoc Ubiquitous Comput.*, vol. 2, pp. 263–277, June 2007.

[19] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[20] M. Horridge and S. Bechhofer, "The OWL API: A Java API for OWL Ontologies," *Semantic Web Journal 2(1), Special Issue on Semantic Web Tools and Systems*, pp. 11–21, 2011.

[21] "JFact DL Reasoner." `http://jfact.sourceforge.net/`. Accessed: January 2015.

[22] Y. Kazakov, M. Krötzsch, and F. Simančík, "The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies," *Journal of Automated Reasoning*, vol. 53, pp. 1–61, 2013.

[23] "The Android Software Development Kit (SDK)." https://developer.android.com/sdk/index.html. Accessed: April 2015.

[24] "Android." https://www.android.com/intl/en_us/. Accessed: April 2015.

[25] Oracle, "Java JDK." http://www.oracle.com/technetwork/java/javase/overview/index.html. Accessed: April 2015.

[26] "Processes and Threads." http://developer.android.com/guide/components/processes-and-threads.html. Accessed: April 2015.

[27] "Sensors Overview." http://developer.android.com/guide/topics/sensors/sensors_overview.html. Accessed: April 2015.

[28] J. W. Lloyd, *Foundations of Logic Programming; (2Nd Extended Ed.)*. New York, NY, USA: Springer-Verlag New York, Inc., 1987.

[29] M. Gelfond, "Answer Sets," in *Handbook of Knowledge Representation* (F. van Harmelen, F. van Harmelen, V. Lifschitz, and B. Porter, eds.), San Diego, USA: Elsevier Science, 2008.

[30] A. Van Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," *J. ACM*, vol. 38, pp. 619–649, July 1991.

[31] M. Schmidt-SchauÃ§ and G. Smolka, "Attributive concept descriptions with complements," *Artificial Intelligence*, vol. 48, no. 1, pp. 1 – 26, 1991.

[32] F. Baader, I. Horrocks, and U. Sattler, "Description Logics," in *Handbook of Knowledge Representation* (F. van Harmelen, F. van Harmelen, V. Lifschitz, and B. Porter, eds.), San Diego, USA: Elsevier Science, 2008.

[33] M. Compton, P. Barnaghi, L. Bermudez, R. GarcÃŋa-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor, "The SSN ontology of the W3C semantic sensor network incubator group," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 17, pp. 25 – 32, 2012.

[34] "Time Ontology in OWL (W3C Working Draft)." `http://www.w3.org/TR/owl-time/`. Accessed: March 2015.

[35] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, pp. 832–843, Nov. 1983.

[36] M. Duerst and M. Suignard, "RFC 3987: Internationalized Resource Identifiers (IRIs)." RFC 3987 (Proposed Standard), see `http://www.ietf.org/rfc/rfc3987.txt`, January 2005. Accessed: April 2015.

[37] B. Motik, P. F. Patel-Schneider, and B. Parsia, "OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition)." `http://www.w3.org/TR/owl2-syntax/`. Accessed: April 2015.

[38] B. Motik, P. F. Patel-Schneider, and B. C. Grau, "OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition)." `http://www.w3.org/TR/owl2-syntax/`. Accessed: April 2015.

[39] I. Horrocks, O. Kutz, and U. Sattler, "The Even More Irresistible SROIQ," in *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006)*, pp. 57–67, 10th International Conference on Principles of Knowledge Representation and Reasoning, AAAI Press, 2006.

[40] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz, "OWL 2 Web Ontology Language Profiles (Second Edition)." `http://www.w3.org/TR/owl2-profiles/`. Accessed: April 2015.

[41] H. Boley, S. Tabet, and G. Wagner, "Design rationale of RuleML: A markup language for semantic web rules," in *International Semantic Web Working Symposium (SWWS)*, pp. 381–402, Citeseer, 2001.

[42] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits, "dlvhex: A Prover for Semantic-Web Reasoning under the Answer-Set Semantics," in *IEEE/WIC/ACM International Conference on Web Intelligence, 2006. WI 2006.* , (Hong Kong), pp. 1073–1074, Dec. 2006.

[43] G. Antoniou and A. Bikakis, "DR-Prolog: A System for Defeasible Reasoning with Rules and Ontologies on the Semantic Web," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, pp. 233–245, Feb. 2007.

[44] D. Nute, "Handbook of logic in artificial intelligence and logic programming (vol. 3)," ch. Defeasible Logic, pp. 353–395, New York, NY, USA: Oxford University Press, Inc., 1994.

[45] V. Vassiliadis, J. Wielemaker, and C. Mungall, "Processing OWL2 ontologies using Thea: An application of logic programming," in *Proceedings of the 5th International Workshop on OWL: Experiences and Directions*, 2009.

[46] M. Şensoy, G. de Mel, W. W. Vasconcelos, and T. J. Norman, "Ontological logic programming," in *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, (New York, NY, USA), pp. 44:1–44:9, ACM, 2011.

[47] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner ," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51 – 53, 2007. Software Engineering and the Semantic Web.

[48] A. Devaraju, S. Hoh, and M. Hartley, "A context gathering framework for context-aware mobile solutions," in *Proceedings of the 4th International Conference on Mobile Technology, Applications, and Systems and the 1st International Symposium on Computer Human Interaction in Mobile Technology*, Mobility '07, (New York, NY, USA), pp. 39–46, ACM, 2007.

[49] O. Lara and M. Labrador, "A survey on human activity recognition using wearable sensors," *Communications Surveys Tutorials, IEEE*, vol. 15, pp. 1192–1209, Third 2013.

[50] L. Chen, C. Nugent, and H. Wang, "A knowledge-driven approach to activity recognition in smart homes," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, pp. 961–974, June 2012.

[51] R. Poppe, "A survey on vision-based human action recognition," *Image and Vision Computing*, vol. 28, no. 6, pp. 976 – 990, 2010.

[52] X. Su, H. Tong, and P. Ji, "Activity recognition with smartphone sensors," *Tsinghua Science and Technology*, vol. 19, pp. 235–249, June 2014.

[53] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, "Using mobile phones to determine transportation modes," *ACM Trans. Sen. Netw.*, vol. 6, pp. 13:1–13:27, Mar. 2010.

[54] H. Xia, Y. Qiao, J. Jian, and Y. Chang, "Using smart phone sensors to detect transportation modes," *Sensors*, vol. 14, no. 11, pp. 20843–20865, 2014.

[55] H. Vathsangam, M. Zhang, A. Tarashansky, A. A. Sawchuk, and G. S. Sukhatme, "Towards practical energy expenditure estimation with mobile phones," in *2013 Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, November 3-6, 2013*, pp. 74–79, 2013.

[56] R. Luque, E. Casilari, M.-J. MorÃşn, and G. Redondo, "Comparison and characterization of android-based fall detection systems," *Sensors*, vol. 14, no. 10, pp. 18543–18574, 2014.

[57] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: Implementing the semantic web recommendations," in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, (New York, NY, USA), pp. 74–83, ACM, 2004.

[58] D. Tsarkov and I. Horrocks, "FaCT++ Description Logic Reasoner: System Description," in *Automated Reasoning*, vol. 4130 of *Lecture Notes in Computer Science*, pp. 292–297, Springer Berlin Heidelberg, 2006.

[59] R. Yus, C. Bobed, G. Esteban, F. Bobillo, and E. Mena, "Android goes Semantic: DL Reasoners on Smartphones.," in *ORE*, vol. 1015 of *CEUR Workshop Proceedings*, pp. 46–52, CEUR-WS.org, 2013.

[60] C. Bobed, fernando Bobillo, R. Yus, G. Esteban, and E. Mena, "Android Went Semantic: Time for Evaluation," in *3rd International Workshop on OWL Reasoner Evaluation (ORE 2014), At Vienna (Austria)*, 2014.

[61] J. Mendez, "jcel: A modular rule-based reasoner," in *In In Proc. of the 1st Int. Workshop on OWL Reasoner Evaluation (OREâĂŹ12*, 2012.

[62] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 Reasoner," *Journal of Automated Reasoning*, vol. 53, no. 3, pp. 245–269, 2014.

[63] Y. Kazakov, "Consequence-Driven Reasoning for Horn $\mathcal{SHIQ}$ Ontologies," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 2009.

[64] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams – a new class of data management applications," in *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pp. 215–226, VLDB Endowment, 2002.

[65] L. Sun, D. Zhang, B. Li, B. Guo, and S. Li, "Activity recognition on an accelerometer embedded mobile phone with varying positions and orientations," in *Proceedings of the 7th International Conference on Ubiquitous Intelligence and Computing*, UIC'10, (Berlin, Heidelberg), pp. 548–562, Springer-Verlag, 2010.

[66] "Pervasive Systems Research Data Sets." `http://ps.cs.utwente.nl/Datasets.php`. Accessed: March 2015.

[67] M. Shoaib, S. Bosch, O. D. Incel, H. Scholten, and P. J. M. Havinga, "Fusion of smartphone motion sensors for physical activity recognition," *Sensors*, vol. 14, no. 6, pp. 10146–10176, 2014.

[68] W. Wu, S. Dasgupta, E. E. Ramirez, C. Peterson, and J. G. Norman, "Classification accuracies of physical activities using smartphone motion sensors," *J Med Internet Res*, vol. 14, p. e130, Oct 2012.

[69] S. Garner, "WEKA: The Waikato Environment for Knowledge Analysis," in *Proc New Zealand Computer Science Research Students Conference*, (University of Waikato, Hamilton, New Zealand), pp. 57–64, 1995.

[70] "An OWL Ontology of Time (OWL-Time)." `http://www.w3.org/2006/time`. Accessed: March 2015.

[71] J. W. Lockhart, T. Pulickal, and G. M. Weiss, "Applications of mobile activity recognition," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, (New York, NY, USA), pp. 1054–1058, ACM, 2012.