A context-aware multi-agent framework for distributed reasoning on Android devices

by

B.V.S Sai Krishna

(Under the Direction of Frederick Maier)

Abstract

This thesis presents the design and implementation of a software framework for distributed reasoning on Android devices. The JPR Agent Library can be used to create agents directly from a Prolog implementation deployable on Android. Each agent contains a knowledge base(KB) and an inference engine. Knowledge sharing is facilitated through a simple message passing scheme. A sample Android application for use in disaster response scenarios is also presented. Additionally, the framework implements a Multi-context system (MCS), MCSs being a powerful theoretical framework for modeling a collection of knowledge sources. Treating each agent KB as a context, the JPR Context Library provides predicates for computing the global well-founded model of the system.

Index words: Context, Context-aware Systems, Multi-agent Framework, Non-monotonic Reasoning, Distributed Reasoning, Rule Based system, Android,Prolog

A context-aware multi-agent framework for distributed reasoning on Android devices

by

B.V.S Sai Krishna

B.Tech., Visvesvaraya National Institute of Technology, 2013

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

© 2015

B.V.S Sai Krishna

All Rights Reserved

A context-aware multi-agent framework for distributed reasoning on Android devices

by

B.V.S Sai Krishna

Approved:

Major Professor: Frederick Maier

Committee: Walter Don Potter
Charles Cross

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2015

Acknowledgments

Foremost, I express my sincere gratitude to my advisor Dr. Frederick W. Maier for his guidance and support over the past two years. I thank you for proof-reading my thesis several times, and bearing with me in general. I also thank Dr. Walter D. Potter, and Dr. Charles Cross for their encouragement and insightful comments.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis presents the design and implementation of a software framework for reasoning in distributed knowledge management systems (DKMS). The framework is developed in Java and Prolog, and while it was designed with mobile Android platforms in mind, it could be deployed on a variety of platforms. In a DKMS, several autonomous computational entities connected over a network facilitate knowledge storage, sharing and retrieval [Akscyn et al., 1988; Bonifacio et al., 2002b,a]. As a part of this thesis, we are interested in systems in which knowledge is formally *represented*, thus allowing for automated *reasoning*.

Context-aware systems can be viewed as a subclass of DKMSs. Context is any information describing the situation of an entity (a person, robot, smart-home etc.) [Dey, 2001]. A system is said to be *context-aware* if it automatically *acquires* an entity's context, *interprets* it and *adapts* itself to it. These systems are usually composed of multiple (connected) devices which perform these operations in a collaborative fashion. Due to the advancements in sensing capabilities of mobile devices (which now typically have GPS, accelerometers, gyroscopes etc.), there has been a growing interest in context-aware systems [Du and Wang, 2008; Hong et al., 2009; Beach et al., 2010; Adomavicius and Tuzhilin, 2011]. However, there are hardly any frameworks that provide a methodology, modeling language, inference engine and communication protocols for developing rule-based context-aware applications on mobile devices [Nalepa and Bobek, 2014]. This research sets out to help fill this gap by incorporating developments in logic-based artificial intelligence (AI).

Knowledge representation and reasoning (KR&R) is a major area of interest in artificial intelligence research [Newell et al., 1959; McCarthy, 1965; Kowalski, 1974; Buchanan and Feigenbaum, 1978; Decker et al., 2000]. The idea of using a fragment of First-Order Logic (FOL) for representation and reasoning led to the development of Prolog [Kowalski, 1974] and other logic programming languages. For logic programs without negation, the semantics (meaning/consequences of a program) are based on the notion of entailment in First-Order logic. But the use of *negation-as-failure* (*not*) [Clark, 1978; Reiter, 1978], a non-classical element, necessitated alternate semantics for logic programs [Van Gelder et al., 1991; Gelfond and Lifschitz, 1988].

For KR&R purposes, the framework proposed in this thesis uses an existing Java implementation of Prolog called Java Prolog Reasoner (JPR) developed at the University of Georgia. JPR was designed especially to make Prolog usable on Android devices. Since context-aware systems often contain multiple connected devices reasoning about an entity, the reasoning module of our framework provides a way of declaratively specifying the knowledge dependencies and is capable of providing a global model of a distributed knowledge base. To facilitate this, we make use of a contextual reasoning framework developed by Giunchiglia and Serafini [1994] and later extended by Brewka et al. [Brewka et al., 2007, 2011; Brewka, 2013].

Contextual reasoning, another area of interest in AI research, is based on the intuition that humans, to reason about a given situation, utilize only a subset of the knowledge they possess [Giunchiglia and Serafini, 1994]. The multi-context system frameworks of Giunchiglia and Serafini [1994; 2001] and their later extensions [Brewka et al., 2007, 2011; Brewka, 2013] formalize this intuition. A *context*, in these theoretical frameworks, is a fragment of knowledge represented in some logic, and the relations between different contexts are captured by *bridge rules*. More recently, the problem of assigning semantics to non-monotonic multi-context systems has been studied [Brewka and Eiter, 2007; Brewka et al., 2011]. However, the majority of the research in this direction is theoretical and only a few implementations

exist. In the ones that do exist, there is hardly any discussion of the applicability for mobile devices. It is important to address this issue because mobile devices are expected to constitute a primary computing platform for the foreseeable future. Our framework for contextual reasoning on Android devices is intended to help fill this research gap.

We use an agent-based approach to perform sensor data acquisition and reasoning without the direct intervention of a human user. Through inter-agent communication, the framework supports knowledge sharing between agents hosted on the same or different devices. The reasoning module is based on the non-monotonic multi-context systems framework of Brewka et al.[Brewka et al., 2011] and is implemented using JPR. In contrast to many of the existing context-aware frameworks, reasoning is performed in a distributed fashion.

1.2 Contributions

The framework is composed of two libraries, one for agents and another for contextual reasoning. The design and implementation of these libraries constitute the primary contributions of this thesis.

1.2.1 The Development of the JPR Agent Library(JAL)

The JPR Agent library is a Prolog library that acts as a bridge between JPR and the Java Agent DEvelopement framework (JADE) [Bellifemine et al., 2007]. Arguably, JADE is the most popular and mature agent development framework available today. The JPR Agent library provides predicates which can be used for creating autonomous *reasoning agents* directly in JPR. Each reasoning agent has its own knowledge base (KB) and inference engine. The KB can be loaded with a set of facts and rules written in Prolog. These may define the actions to be performed by an agent such as sensor data collection[1], context enhancement (for example, using accelerometer data to infer that a user is running) etc. Applications

[1]To interact with sensors directly from JPR, a separate sensor library is available.

can periodically query different agents and trigger corresponding changes. A communication module provides predicates for knowledge sharing among agents.

This Prolog based multi-agent system can be used for a wide range of applications including:

- Adaptive Systems and Smart spaces: Detecting the context of a user is highly useful for adaptive applications including navigation, gaming, smart homes and others.
- Health Care: Context-aware systems utilizing wearable devices are expected to be highly useful in recording and analyzing patient data. The problem of fall-detection in elderly people has been widely discussed [Nyan et al., 2008; Kangas et al., 2007; Doukas et al., 2007].
- Response monitoring in post-disaster scenarios. The discussion of the agent library in chapter 3 is built around an instance of disaster response.

1.2.2 JPR Context Library(JCL)

The JPR context library (JCL) provides predicates which can be used to compute the well-founded [Van Emden and Kowalski, 1976] consequences of a distributed knowledge management systems (DKMS). This library extends the JPR Agent Library, treating each agent KB as a context and providing extra predicates for computing the consequences of a multi-context system. The well-founded model computation is performed in a distributed fashion with a central coordinating agent. Both JAL and JCL can be used to develop applications for Android devices or machines running Java.

1.3 Outline

The thesis is composed of four chapters. In the next chapter, we cover some preliminary background on three topics, namely multi-agent systems, context-aware systems, and logic-based artificial intelligence (AI), laying special focus on contextual reasoning. In AI, context-related

research has a long history with significant contributions from pioneers such as McCarthy [1989], Giunchiglia [1994], and others. The research on context-aware systems, in contrast, is relatively young and shows promise due to the advances in sensing capabilities of mobile devices. We are interested in the merging of the two.

In chapter 3, we discuss the design, implementation, and limitations of JAL. We leverage existing work in agent-related research, specifically the Java Agent DEvelopment (JADE) framework [Bellifemine et al., 2007]. JADE has been widely used in the past for both research and industry-level applications [Bogdanova et al., 2014; Su and Wu, 2011; Zhao et al., 2007; Bădică et al., 2006]. Having described the agent library in detail, in chapter 4, we move on to discussing the implementation of a multi-context system [Brewka et al., 2011, 2007; Brewka, 2013] using an agent-based middleware. The non-monotonic multi-context systems framework proposed by Brewka et. Al [2007; 2007; 2011; 2013] is a prominent framework proposed for contextual reasoning. It allows heterogeneous knowledge sources (logics) and supports non-monotonic reasoning. In chapter 5, we summarize the work presented, limitations and future scope of this research.

Chapter 2

Agents, Context-Aware Systems and Logic-based AI

2.1 Introduction

This thesis presents the design and implementation of a context-aware multi-agent framework. In this chapter, we provide background information on three research areas that the remainder of the thesis draws upon: agents, context-aware systems, and logic-based AI.

In general, we take an agent-based approach to implement the autonomous computational entities that constitute a distributed knowledge management system (DKMS). In section 2.2, we provide a brief overview of agent-related research in AI. We discuss Jennings and Woolridge's definition of an agent [Wooldridge and Jennings, 1995], various types of agent architectures, agent programming and communication languages.

The literature on DKMSs is relatively sparse. So, in section 2.3, we present a review of context-aware systems which can be viewed as a subclass of DKMSs. We discuss Knappmeyer's [2009] definition of *context*, various types of context information and the requirements that a context-aware framework must meet. Though context aware systems and KR&R generally constitute distinct areas of research, there has been work on using KR&R techniques including ontological and rule-based approaches for reasoning (interpreting) about context information [Etter et al., 2006; Henricksen and Indulska, 2006; Jaroucheh et al., 2011; Knappmeyer et al., 2009; Luqman and Griss, 2010; Moawad et al., 2013].

The primary aim of this chapter, however, is to provide necessary background on various concepts in logic-based AI, especially the semantics of logic programs [Van Gelder et al., 1991; Gelfond and Lifschitz, 1988] and the multi-context systems frameworks of Giunchiglia and Serafini [1994], and Brewka et al. [2007; 2011]. Precisely, we use an agent-based approach

to model a rule-based context-aware system and allow for distributed reasoning based on the multi-context systems framework.

2.2 Agents

One of the primary goals of the field of Artificial Intelligence (AI) is to create intelligent artifacts. The *agent* is a fundamental abstraction AI researchers have developed to achieve this goal. Amongst the several definitions of the term '*agent*' available in the literature, the one most relevant to the scope of this paper is given below [Wooldridge and Jennings, 1995].

Definition 2.2.1. *An agent is a software component with the following properties:*

- *Autonomy: Agents operate without the direct intervention of humans or other agents. They execute complete control over their actions and internal state.*
- *Social ability: Agents communicate, at least with other agents, if not humans.*
- *Reactivity: Agents sense and react to changes in their environment.*
- *Pro-activeness: An agent should be capable of exhibiting goal directed behaviors.*

Based on its current knowledge and percepts, an agent architecture must enable an agent to infer which actions it should perform and its future states. A broad classification of agent architectures is given below [Wooldridge and Jennings, 1995].

- Deliberative: Deliberative architectures are based on the Physical Symbol System Hypothesis [Simon, 1969] which states that a symbolic representation of the world is both necessary and sufficient to generate intelligent action. Therefore deliberative agents possess an explicit representation of their environment as physical symbols. Actions to be executed are decided by some symbolic manipulation process.
- Reactive: A reactive architecture, on the other hand, does not involve an explicit representation of the world. The subsumption architecture [Brooks, 1991] is a classic

example. In Brook's architecture, an agent's sensory inputs are directly coupled to its actions. Therefore, the response time for reactive agents is very low. However, due to the lack of an internal representation, agents usually fail to learn from their experience.

- Hybrid: Hybrid architectures tend to provide a balance between the reactive and deliberative approaches. The Procedural Reasoning System of Georgeff and Lansky [1987] falls into this category. The knowledge of an agent is decomposed in to so called 'knowledge areas' where each knowledge area represents the beliefs, desires and intentions of an agent in first-order logic. A *reactive* mapping from sensor data inputs to a particular knowledge area is provided.

The above classification is not exhaustive. Badica et al. [2011] survey several existing frameworks for rule-based agent systems, and identify some architectures which lie at reactive-hybrid or hybrid-deliberative boundaries.

2.2.1 Agent Programming Languages

Agent programming languages are special purpose programming languages which provide support for representing various aspects of agency such as beliefs, roles and plans etc. in a multi-agent setting. Bordini et al.[2006] survey the existing agent programming languages and platforms for multi-agent systems. Some languages take an imperative approach [Howden et al., 2001] while others take a declarative approach [Thielscher, 2005; El Fallah-Seghrouchni and Suna, 2004; Li, 2001]).

Objects in an Object-oriented programming language provide a natural abstraction. However, agents and objects cannot be equated unless the latter executes control over its own thread of execution (autonomy) [Jennings et al., 1998]. The following are some benefits of taking a declarative approach [Baldoni et al., 2010]:

- Various aspects of agency such as knowledge, beliefs and goals can be formally represented.

- The syntactic elements of interaction protocols for multi-agent systems can be easily implemented.
- Agents can be made proactive using a goal-directed reasoning processes.

Hybrid languages, which implement the best features from both have also been developed [Hindriks et al., 1999].

2.2.2 Agent Communication Languages

To perform a collaborative task, the agents constituting a multi-agent system must be able to communicate with each other in a meaningful fashion. Agent communication languages are specifically designed for this purpose. They specify a structure for the messages exchanged between agents. To promote a fruitful interaction, understanding the structure and content of a message is not sufficient. It is necessary for a sending agent to specify its *intention* in sending a particular message [Searle, 1969].

For example, consider the FIPA agent comunication language (FIPA-ACL) Fipa [2002a] which is a standard agent communication language in the field [Bellifemine et al., 2007]. The elements of a FIPA-ACL message structure are shown in Fig. 2.1. A FIPA-ACL message bears several parameters of which the *performative* (type of communicative acts in Fig. 2.1) is mandatory. It indicates the intention of the sender and can take values such as request, inform, agree, cancel etc. The *content* of the message refers to the actual communicative message being passed. The agents exchanging messages are identified using the *sender*, *receiver* and *reply-to* parameters. The characteristics of the message such as its language and encoding can also be specified. The language in which the message is encoded is called the content language of the message. All the elements of a FIPA message are shown in Fig. 2.1.

2.2.3 Foundation for Intelligent Physical Agents (FIPA)

The literature on agent systems does not indicate any clear consensus on the type of architecture, or programming language suitable for implementing agent systems, and the present

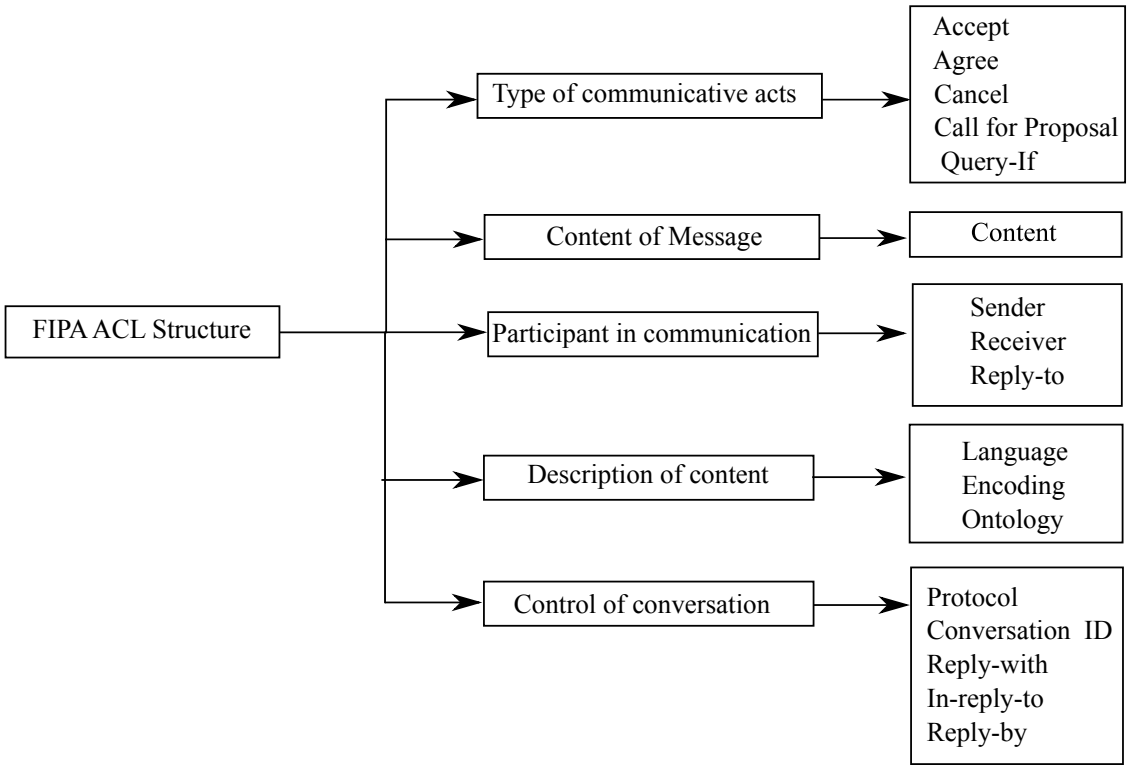

Figure 2.1: The elements of a FIPA-ACL message specification

systems differ from each other along several dimensions. To promote interoperability between existing agent systems, the Foundation for Intelligent Physical Agents (FIPA) [1], now an IEEE Computer Society standards organization, was set up in 1996. FIPA published 25 specifications covering an abstract architecture for an agent platform [FIPA, 2002], agent management services [Fipa, 2002b], structure of agent messages [Fipa, 2002a] etc. FIPA's methodology for compiling these specifications is explained in a work by Posald [2007]. In our opinion, these are among the best specifications available in the field of agent systems. We do not delve any further into these specifications here, but in chapter 3, a detailed description

[1] http://www.fipa.org

of the Java Agent DEvelopement framework (JADE) [Bellifemine et al., 2007], which is fully FIPA-compliant, is provided.

2.2.4 Agent Frameworks

In recent years, several agent software development frameworks have been implemented [Bellifemine et al., 2001; Morley and Myers, 2004; Winikoff, 2005; Chen et al., 2006; Santi et al., 2011]. They provide generic functionalities of an agent system, thereby, allowing a developer to focus on application oriented issues. For example, JADE uses Java as the agent programming language and FIPA-ACL as the agent communication language. Various elements of the FIPA-abstract architecture [FIPA, 2002], FIPA-agent management system [Fipa, 2002b], communication and interaction protocols are implemented. Developers can embed application-specific code into agents and launch them. In chapter 3, we will discuss more about the JADE framework and the way we have integrated it with JPR to create a context-aware system using an agent-based middleware.

2.3 Context-Aware Systems

Context-aware systems are a general class of mobile systems that can sense their physical environment and adapt their behavior accordingly. Dey [2009] defines context as "any information that can be used to characterize the situation of an entity." The Active Badges System [Want et al., 1992] is arguably one of the earliest context-aware systems developed. It is essentially a call forwarding system which, in a large office-like environment, forwards phone calls to the phone nearest to the intended receiver. The users are not stationary and the location of each user is determined using a RFID tag attached to him/her. In the following years, various context-aware systems were developed in different domains [Abowd et al., 1997; Abowd, 1999; Cheverst et al., 2000; Salber et al., 1999].

In this section, we are going to discuss context-aware systems in more detail especially the various types of context data, reasoning about context information and context modeling.

Our aim is not to be exhaustive but provide a rough idea of context-aware systems and research in this area. We also point to several articles which discuss this topic in more detail.

2.3.1 Context

In general any information about the user that an application is interested in can be termed as context. More so if the information changes during the lifetime of an application. Knappmeyer et al. [2009] identify the following types of context:

- Spatial: Information related to the geographical location of a user such as GPS co-ordinates, proximity to a building etc.
- Temporal: Absolute time of the day or time relative to the occurrence of an event.
- Device: Device information such as sensor data, processing power etc.
- Network: Wireless access point, bandwidth, throughput etc.
- Activity: Information related to tasks being performed by a user such as running, reading, sleeping etc.
- Other: Other types of context include emotional, mental, interaction etc.

They also provide an extended definition of the term 'context':

> *"Context is any information that provides knowledge and characteristics about an entity (a user, an application/service, a device, or a spatially bound smart place) which is relevant for the interaction between the entities themselves and with the digital world. Context can be categorized as being static, dynamic and rapidly changing." [Knappmeyer et al., 2009]*

This study does not investigate in to the mental or emotional types of context. For us, context is any information obtained directly from sensors on Android devices or by performing some sort of inferencing on it.

2.3.2 Context Modeling and Reasoning

It seems to be a generally held view in the area of context-aware systems that the raw context data obtained from sensors cannot be used directly for driving intelligent actions. A high-level knowledge representation model of the context data called a *context model* is required. It promotes interoperability between different layers or modules of a context-aware system by serving as a common understanding of an entity's context.

A survey of context modeling techniques and requirements is provided by Bettini et al. [2010] and Strang et al. [2004]. The context models can be classified into: key-value Pairs [Schilit et al., 1994]; graphical models [Henricksen and Indulska, 2006; Bui et al., 2001; Riboni and Bettini, 2011]; ontological models [Chen et al., 2005; Saleemi et al., 2011]; and rule-based models. A knowledge base and an inference engine are the two main components of a rule-based system. The knowledge base contains a set of rules (and facts) about the target entity. The inference engine periodically checks the fact base and fires the rules accordingly. Rule-based systems, by allowing the addition and deletion of rules and facts, facilitate the development of dynamic systems. Though there are popular rule based environments such as CLIPS [Giarratano et al., 1993], JESS [Friedman-Hill, 2003], Drools [Bali, 2009] etc. their application to context-aware systems has been limited [Daniele et al., 2007].

Moreover, to simplify the development of context-aware applications, some researchers in the past have proposed specific software frameworks for them [Knappmeyer et al., 2009; Loke, 2004; Luqman and Griss, 2010; Saleemi et al., 2011]. Dey [2009] proposed a set of design requirements that a context-aware framework should fulfill to ease the development of context-aware applications. These requirements tend to simplify the task of accessing and utilizing context information, allowing an application developer to focus on the core functionalities of the application. These requirements are listed below.

1. Separation of Concerns: The task of context acquisition i.e. acquiring sensor data by directly accessing the respective sensors must be abstracted away from the developer.

Different sensors may have different protocols, and behaviors. A framework must provide a simple means of obtaining sensor data and handling it in more or less the same way as user input.

2. Continuous Context Acquisition: The program components responsible for context acquisition should be running independently of the program itself, allowing it to query when necessary. These components should be persistent and available at all times.

3. Interpretation: More often than not, the low level context information is of little importance to an application. So the context information should be abstracted (at different levels) before it is handed over to an application. Also, the application should still be able to query the low level information, if necessary.

4. Communication: Generally a context-aware application derives inputs (sensor data) from several sources which are distributed in a physical environment. The framework should provide a suitable communication mechanism for interaction between the application and sensors.

5. Context Storage: The context history can be used for predictions, analyzing user behaviour and providing user recommendations. So the architecture should provide means for context storage.

6. Resource Discovery: Frameworks should provide a resource discovery module that allows applications to find different sensors, services they offer and protocols to access them.

Our framework uses the JPR sensor library [Cline, 2015], developed at University of Georgia as a part of a separate project, for sensor data collection. This satisfies requirement 2. We use the elements of JADE framework to satisfy requirements 4,5, and 6 above. Further details of the framework are presented in chapter 3. In the following section, we turn to the more theoretical aspects in logic-based AI that this project builds upon.

2.4 Logic-based AI

The importance of assigning AI programs a context was first highlighted by McCarthy [1987]. Often, AI programs had to be restructured at the data structure level if they had to be used for reasoning about a situation which is more *general* than the one they were originally designed for. This problem is referred to as the *problem of generality*. Representing the 'context' of a program as a formal object, and using non-monotonic reasoning was proposed as a possible solution to this problem. In the following years, a considerable amount of literature was published, extending this proposal and some alternate ways of approaching this problem [McCarthy, 1993; Guha, 1991; Ghidini and Giunchiglia, 2001; BuvaE and Mason, 1993; Bouquet et al., 2003; Brewka and Eiter, 2007; Bikakis and Antoniou, 2010b; Brewka et al., 2011; Brewka, 2013]. Following is a brief review of literature published in this regard [Przymusinski, 1989; Minker, 1993; Apt and Bol, 1994].

In 1959, Newell and Simon [1959] proposed the General Problem Solver (GPS). The idea was to create a generic solver to solve any problem that is *formally* represented. General Problem Solver was successful in solving some toy problems but suffered from combinatorial explosion when scaled to real-world problems. However, their idea of separating the solver and the problem heavily influenced the research in logic-based AI, in the following years.

Around the same time, McCarthy [1963] proposed the use of first order logic to represent knowledge in AI programs. In the early 1970s, Colmeraner and Kowalski [1973; 1974] invented the Prolog programming language which, arguably, remains the dominant declarative language in AI research. The fundamental aim of the declarative paradigm [Kowalski, 1974], is summarized below [Kunen, 1987]:

> *"The user should be able to understand the semantics just by 'logic'; not by a detailed understanding of the implementation of the interpreter. It is really this key fact that separates logic programming from more conventional procedural forms of programming."*

However Marvin Minsky, argued that systems implemented using classical logic cannot adequately imitate human-like reasoning [Minsky, 1974] . This is primarily due to the monotonic nature of classical logic i.e. addition of new knowledge to an existing program cannot defeat any of the existing conclusions. As a result, several non-monotonic formalisms including Default Logic [Reiter, 1980], circumscription [McCarthy, 1980], defeasible logic [Nute, 1994], and several others were defined during this period.

In 1987, McCarthy [1987] asserted that the problem of generality had not been solved. He cited two indications of this.

- AI programs often need to be re-written when new knowledge is added.
- A suitable method of encoding common sense knowledge, that we humans often rely on, is not found yet

He proposed that introducing a formal notion of *context* in AI programs, and combining it with circumscription [McCarthy, 1980] to handle non-monotonicity could be a possible solution to this problem. An intuitive meaning of context, as a situation in which certain axioms hold was provided [McCarthy, 1980]. Contexts can be ordered in terms of *generality*. If a context C_1 is more general than a context C_2, then all the axioms that hold in C_2 also hold in C_1.

Based on the intuition that reasoners, like humans, need not use all the knowledge they possess, the multi-contexts systems framework [Giunchiglia and Serafini, 1994; Ghidini and Giunchiglia, 2001] was proposed. Context is the partial knowledge that a reasoner can use to solve a set of goals. In the following years, this notion of context has received more attention and the recent works including ours are based on this notion of context.

As far as the recent developments in regard to the MCS framework are considered, there are two major camps. The first focuses on extending the Multi-Context Systems framework to support heterogeneous non-monotonic cases. By heterogeneous, we mean the ability to support different types of logics in a single MCS. The second camp is focused on query

evaluation in a distributed setting, where each local theory contains strict and defeasible rules (defeasible logic). We discuss both the approaches in the following section. Fortunately, both the camps have discussed, to some extent, implementation of their frameworks including algorithms, complexity and initial testing on toy problems, which are elaborated upon in the following chapters.

2.4.1 Non-monotonic Logics and Semantics of Logic Programs

In this section, we provide a brief overview of logic programming and briefly discuss their semantics (meaning). A logic program is a set of rules written using a first-order logic like syntax. Assuming that the reader has a knowledge of syntax of Prolog, we skip the discussion on the syntax of logic programs and rather focus on their semantics. Logic programs which do not involve any form of negation (*not* or $\neg$) are called definite logic programs.

Definition 2.4.1. *A definite logic program, say P, is a set of clauses of the form*

$$A \longleftarrow A_1, A_2, A_3....A_n, \text{ where } A_1...A_n \text{ are atomic.}$$

The syntactic element connecting the head of the rule (':-' in Prolog, '$\Leftarrow$ ' in some other logic programming languages) and its body, is treated as a FOL implication. There is a general consensus on the semantics of definite logic programs[2]. We briefly discuss it here.

Example 2.4.1. *The following program S,*

$$p(a).$$
$$p(b).$$
$$q(x) :- p(x).$$

is an instance of a definite logic program with three rules (the first two, without bodies).

Definition 2.4.2. *For a logic program P, the Herbrand base B_P is defined as the set of all ground atoms formed from the predicates, constants and function symbols in P.*

[2]The semantics of definite logic programs is discussed in many logic programming textbooks

If function symbols are present, an infinite number of ground atoms can be formed in a recursive fashion. Hence the Herbrand base of a program involving them is infinite.

Example 2.4.2. *For program S above, the Herbrand base is* $\{p(a),\ p(b),\ q(a),\ q(b)\}$.

Definition 2.4.3. *A Herbrand interpretation of a program P is any subset of the Herbrand base* B_P *. A Herbrand interpretation of a program P is a model of P iff it is a model of all the clauses in P. An interpretation I is a model of clause C if for each ground instantiation of C, the head (C) is an element of I if all the elements in body(C) are also elements of I.*

Example 2.4.3. *For program S above, {p(a), p(b), q(a), q(b)}; {p(b), q(a)}, {}, are all valid interpretations. But only the first interpretation is a model.*

Definition 2.4.4. *The intersection of all the Herbrand models of a program P is its minimal Herbrand model. The minimal Herbrand model constitutes the semantics of definite logic programs.*

One way to compute the minimal Herbrand model is by using the immediate consequence operator γ_P which, for an interpretation I, is defined as

$$\gamma_P(I) = \{head(r) \mid rule\ r\ \in\ B_P\ and\ body(r)\ \subseteq\ I\}$$

Starting with an empty interpretation and applying the γ_P iteratively, a fixed point is reached such that γ_P (I(m+1)) = γ_P (I(m)), where m is the number of iterations. It has been showed that this unique fixed point of a program P equals the minimal Herbrand model.

So-called normal logic programs extend definite programs by allowing negation-as-failure. The notion of *failure-to-prove* is semantically problematic. For example,

Example 2.4.4. *Consider the program P = { p :- not(q). }. In Prolog, the intuitive meaning of not(q) is that not(q) is true if q is not provable. Since q is not provable, the program returns true when queried for p. If q is added as a fact, a query for p no longer succeeds.*

Therefore, with normal logic programs, it is possible that the conclusions made by a program are no longer valid, after adding new knowledge to it (non-monotonicty).

Example 2.4.5. *Moreover, consider the following Prolog program with two rules,*

$$P = \{p :- \ not \ q. \ , \ q :- \ not \ p.\}$$

When queried for either literal, a Prolog program goes in to an infinite loop. This behavior is comprehensible only if one knows the way negation as failure *works* in Prolog. But having to deal with the way a program operates conflicts with the basic idea of declarative programming. To overcome this problem, various semantics with the following common goals were proposed [Apt and Bol, 1994]:

1. According to any given semantics for normal logic programs, the consequences of a program, without negation should be equivalent to the minimal Herbrand model.

2. To find a *reasonable* canonical model for programs involving negation as failure. Each semantics differs from others in the way they characterize what is *reasonable*.

Ideally, by adopting one of these semantics, the developer must be able to work with a logic program without having to know the way *not* is handled by the inference engine i.e. restoring the declarative nature.

The most popular semantics for logic programs are the stable model/answer-set semantics proposed by Gelfond and Lifschitz [1988], and the well-founded semantics proposed by Van Gelder et al. [1991]. Due to reasons of space, we are not going to discuss the stable model semantics in detail. But of interest to us is the Gelfond-Lifschitz operator which is also usared in the computation of the well-founded semantics, explained later.

Definition 2.4.5. *(Reduct) The reduct of a normal program P with respect to an interpretation I is obtained by,*

1. *deleting each rule r such that for any term p :* $\mathsf{not(p)} \in body\,(r) \mathsf{and}\ \mathsf{p} \in \mathsf{I}$

2. *deleting all the other not q from rules of P.*

Example 2.4.6. *Consider the normal logic program,*

$$P = \{a.,\ b :- not\ a.,\ c :- not\ d.,\ e :- c.\}.$$

The reduct of the program P with respect to interpretation I = {} is {a, c, e:-c.}.

Definition 2.4.6. *The Gelfond-Lifschitz operator $\Upsilon(P^I)$ maps a program P to the closure of the consequences of its reduct with respect to an interpretation I.*

The closure of a definite program is equivalent to its minimal Herbrand model. For the above program P, $\gamma(P^I) = \{a,c,e\}$.

Definition 2.4.7. *An interpretation I is a stable model of a normal logic program P iff $\gamma(P^I)$ is equivalent to I.*

For the program P above, I = {a,c,e} is an interpretation because $\gamma(P^I)$ ={a,c,e}. Stable models constitute two-valued models of normal programs. If one thinks of normal programs as a set of constraints, each stable model constitutes a way of satisfying the constraints. However, note that there is no iterative procedure to compute all the answer sets of a given program.

The well-founded model semantics [Van Gelder et al., 1991] offers a three valued interpretation and consequently, three distinct sets of ground terms : the well-founded, unfounded and undefined. According to WFS, all the literals in the well-founded set are true and those in the unfounded set are false. The members of the undefined set are neither true nor false.

Definition 2.4.8. *The well-founded model of a logic program P, represented wfm(P), is the tuple $\langle T, U \rangle$ such that,*

1. *$T = lfp\ (\gamma_P^2)$*
2. *$U = B_P$ - $gfp(\gamma_P^2)$*

The well-founded model can be calculated using a fix-point procedure similar to the one mentioned in case of definite logic programs. The only difference in this case is that the fix

n	$\gamma_P(I)$
0	{}
1	p,q
2	{} (lfp)
3	p,q (gfp)
4	...
5	...

point alternates between the lfp (least fixed point) and the greatest fixed point (gfp). Note that the operator γ^2 is monotonic.

Example 2.4.7. *Consider the following program with two rules,*

$$P = \{p :- \ not \ q. \ , \ q :- \ not \ p.\}$$

The computation of the well-founded model proceeds as shown in table 2.4.7:

The answer set semantics and the stable model semantics, stick to a two valued interpretation where there is no concept of 'undefined' but all the atoms are either true or false according to an interpretation. The discussion of semantics for extended logic programs, involving both *not* and classical negation, is out of the scope of this thesis.

2.4.2 Multi-Context Systems and Defeasible Contextual Reasoning

So far, we have discussed the semantics for a single logic program. But often, the knowledge pertaining to a situation or an entity may be distributed i.e. contained in multiple logic programs. For example, consider a smart-home environment monitored by three logic-based software agents: one each for location monitoring, activity detection and temperature control. Also assume that each of them is equipped with sensors and a set of rules (logic program), allowing them to infer higher-order context information about their environment. The agent responsible for location monitoring may conclude that the user is in the kitchen, or on a

treadmill etc. This information can be used by the activity recognizer agent to detect if the user is walking or running. The temperature controlling agent, based on information from the other two agents (say "the user is running on a treadmill"), can trigger increase/decrease the temperature of the smart-home. To actually implement such a system,

- A method of representing rules involving terms from different programs must be provided.
- A formal semantics for the whole system (encompassing all the programs) must be provided.

The multi-context system framework [Giunchiglia and Serafini, 1994], precisely, serves this purpose. Here we discuss the most recent and complete version of this framework, the heterogeneous non-monotonic Multi-Context Systems [Brewka et al., 2011].

Definition 2.4.9. *[Brewka et al., 2011] A logic $L = (KB_L, BS_L, ACC_L)$ is composed of the following components:*

- *KB_L is the set of well-formed knowledge bases of L. We assume that each element of KB_L is a set*
- *BS_L is the set of possible belief sets,*
- *$ACC_L : KB_L \rightarrow 2^{BS_L}$ is a function describing the semantics of the logic by assigning to each element of KB_L a set of acceptable sets of beliefs.*

For example, consider normal logic programs under well-founded semantics, defined over a set of literals Σ,

- KB is the set of normal logic programs over Σ
- BS is the set of sets of atoms over Σ (each set constituting an interpretations)
- ACC(kb) = wfm(kb), the acceptable belief set of a *kb* is the singleton set containing the well-founded model of kb.

Definition 2.4.10. *[Brewka et al., 2011]Let $L = \{L_1, ..., L_n\}$ be a set of logics. A L_k-bridge rule over L, $1 \leq k \leq n$, is of the form $s \leftarrow (c_1 : p_1), ..., (c_j : p_j), not(c_{j+1} : p_{j+1}), ..., not(c_m : p_m)$ where $1 \leq r_k \leq n, p_k$ is an element of some belief set of L_{rk}, and for each $kb \in KB_k$, it holds that $kb \cup \{s\} \in KB_k$.*

The interconnections between the different knowledge bases are modeled using bridge rules. The head of a bridge rule is always a local literal, while the body may contain local or foreign literals.

Definition 2.4.11. *[Brewka et al., 2011] A multi-context system $M = (C_1, \ldots, C_n)$ consists of a collection of contexts $C_i = (L_i, kb_i, br_i)$ where $L_i = (KB_i, BS_i, ACC_i)$ is a logic, kb_i is a knowledge base (an element of KB_i), and br_i is a set of L_i-bridge rules over $L_1, ..L_n$.*

Example 2.4.8. *A multi-context system $M = \{C_1, C_2, C_3\}$, where all the contexts are normal logic programs under well-founded semantics (same L_i) is given below.*

$$kb_1 = \{a.,\ b.,\ c \leftarrow d.\} \quad and\ br_1 = \{\}$$

$$kb_2 = \{r.\} \quad and\ br_2 = \{p \leftarrow 1 : a.,\ q \leftarrow 3 : x.\}$$

$$kb_3 = \{y.\} \quad and\ br_3 = \{w \leftarrow 2 : q.,\ x \leftarrow w.\}$$

Any set of literals in kb_1 constitute its belief set. The only acceptable belief set in kb_1 is its well-founded model $\{a,b\}$. Similar relations hold for other KBs.

The semantics of a multi-context system are defined in terms of *belief states*, where a belief state is a sequence S = $\langle S_1, S_2, ...S_n \rangle$ such that each S_i is an element of BS_i. Intuitively, a belief state is a combination of belief sets, one from each context.

Definition 2.4.12. *[Brewka et al., 2011] A belief state $S = \langle S_1, ..., S_n \rangle$ of a multi-context system M is an equilibrium iff $S_i \in ACC_i(kb_i\ U\ head(r) | r \in app(br_i, S), 1 \leq i \leq n.$, where $app(br_i, S)$ refers to an applicable (all the elements in its body are true) bridge rule.*

Intuitively, a belief state is an equilibrium if each of the component belief sets are acceptable, given others. For the system in Example 2.4.8, S = $\{\{a, b\}, \{p, q, r\}, \{w, x, y\}\}$ is an equilibrium. Moreover, Brewka and Eiter [2007] also show that computing the equilibrium of a MCS under well-founded semantics (WFS), is polynomial in time with respect to the size of the program. It is this key fact that mainly motivates the use of WFS for this project.

2.4.3 Defeasible Contextual Reasoning

Moving on, the other camp we mentioned earlier, i.e. Antonio's group mainly focuses on developing algorithms for distributed query evaluation. Unlike the MCS framework, where contexts could be defined in different logics, the focus here is on extending defeasible logic [Nute, 1994] to a distributed setting. Currently, there are several versions of defeasible logic and corresponding semantics [Maier, 2007; Maher, 2002; Governatori et al., 2004; Maier and Nute, 2010]. Due to reasons of space, here we only discuss the syntax of defeasible logic.

According to Nute [1994], atomic formulas in defeasible logic are defined in the same way as in FOL. A *literal* is any atomic ϕ formula or its negation. ϕ and $\sim \phi$ are complements of each other. The following types of rules are defined (for any set of literals S and a literal p):

- *Strict rules*, which take the form S $\rightarrow$ p
- *Defeasible rules*, which are written S $\Longrightarrow$ p
- *Undercutting defeaters*, which take the form S $\sim>$ p

The strict rules can never be defeated, i.e. if the antecedent of a strict rule is true, then the consequent must be true. The defeasible rules, on the other hand, are more like rules of thumb. Undercutting defeaters can be used to specify exceptions to defeasible rules, but cannot be used as a rule of inference themselves. For example, from S $\sim>$ $\neg p$, we cannot directly infer $\neg$p, but other ways of concluding p can be blocked. A precedence relation among defeasible rules and under-cutting rules is defined. This ordering is used to resolve

conflicts. Though the specifics depend on the particular logic, a conclusion P is (defeasibly) derivable if

- P is a fact, or
- There is an applicable strict or defeasible rule for P, and all rules for ¬P are blocked or weaker than an applicable strict or defeasible rule for P.

For contextual reasoning, Bikakis and Antonis [2010a], use a restricted form a defeasible logic, one with out facts, defeaters and superiority relation among rules.

Definition 2.4.13. *A context is a tuple of the form* (Vi, Ri, Ti)*, where*

- V_i *is a set of positive and negative literals,*
- R_i *is a set containing local and mapping rules. The local rules can be strict or defeasible where as the bridge rules are defeasible.*
- *and* T_i *is a total preference ordering on C where C is a set of contexts. In case of conflicting conclusions from different contexts, a given context accepts the conclusions from the context higher in the preference ordering. Intuitively, preference ordering is the ordering of belief a context has on others.*

In contrast to Brewka's work, this can be viewed as a framework for query evaluation, instead of model building. The distributed query evaluation algorithm returns the truth value of a literal issued to a local theory i.e. a context C_i, in a MCS = $\{C_1, C_2, C_3.....C_n\}$ [Bikakis and Antoniou, 2010a]. Consider a query for literal p_i at C_i, it proceeds in four main steps. In the first step, the algorithm checks if the given literal or its contradiction (p_i or $\neg p_i$) can be proved from the strict rules in C_i. The algorithm terminates in either case, returning the respective truth value. If the literal cannot be proven locally, it locally checks for rules with p_i as the consequent. For each foreign literal in the body of such rules, a query is initiated to the corresponding context. If all the elements in the body of a rule are computed to be

true, then it is *applicable*. A rule is *unblocked*, if any element in its body is either true or undefined. Also the following sets of foreign literals are formed: one containing the literals appearing in most preferred chain of applicable rules for p_i (SS_{p_i}) and another containing the literals appearing in most preferred chain of unblocked rules for p_i (BS_{p_i}). The same procedure is repeated for $\neg p_i$. The algorithm returns true if there is no unblocked rule for $\neg p_i$, or if SS_{p_i} is computed to be stronger than BS_{p_i}. The algorithm returns false if there is at least one applicable rule for $\neg pi$ and BS_{p_i} is not stronger than SS_{p_i}. In any other case, the truth value is undefined.

2.5 Conclusion

In this chapter, we have provided a brief overview of agents, context-aware systems and contextual reasoning. We have discussed the logic-based foundations of this thesis, namely the semantics for normal logic programs and the multi-context system framework in a detailed fashion. The following chapters elaborate on the original contributions of this thesis: the agent library and the multi-context system framework.

Chapter 3

JPR Agent Library

3.1 Introduction

JADE is a FIPA-compliant agent development framework [Bellifemine et al., 2007]. The JPR Agent Library (JAL) acts as a bridge between JPR and JADE, thus enabling the creation and maintenance of agents through JPR. In this chapter, a brief overview of JADE and JPR is provided. Using disaster management as a motivating scenario, the design, implementation, and limitations of JAL are discussed. In chapter 4, we explain how JAL helps us realize the primary goal of this project, namely the implementation of a multi context system on Android devices.

3.2 Motivating Scenario

In this section, we present disaster response as a motivating scenario for the agent library. This scenario will be used throughout the chapter to illustrate various functionalities of JAL. At the end of this chapter, we present an Android application developed using this library, that could possibly be used in real-world scenarios.

Disaster management involves creating plans to cope with a disaster. There are usually five phases of disaster management: prevention, mitigation, preparedness, response and recovery[1]. The first three phases involve environmental planning and design, training emergency responders and other activities. The response phase involves search and rescue operations, providing shelter, food and medical care to the victims, and managing additional

[1]http://www.fema.gov/media-library-data/20130726-1914-25045-8516/final_national_response_framework_20130501.pdf

resources. During the recovery phase, efforts are made to restore the normal functioning of an affected area (rebuilding infrastructure).

For present purposes, we are interested in the response phase. The personnel involved during this phase are typically *emergency responders* and *volunteers*. The former includes fire marshals, 911 personnel, or members of humanitarian organizations who are professionally trained for disaster response activities. There exists a predefined information flow from the personnel working on the ground level to the ones at the top-most level where overarching decisions regarding the response activities are usually taken. Volunteers are the people, mostly from surrounding areas, who gather after the disaster in an ad-hoc fashion. They usually outnumber the emergency responders. Since volunteers do not often have professional training, collecting data from them, coordinating (volunteers or volunteers-responders) and planning a disaster response is quite challenging.

To simplify this task, we propose a collaborative knowledge management system (CKMS), a software system which could serve the following purposes:

- Receive disaster-related data from various volunteers and responders (via Android devices)
- Represent the data in a suitable form for automated reasoning and
- Respond to formal queries for disaster information

The disaster related data could be the number of persons injured/dead, individual details of victims(name, age, sex), the amount of food supplies required etc. Disaster control centers (DCC), which are setup soon after a disaster has occurred, host high-level authorities, decision makers and several other personnel who plan/monitor the response activities. In the following sections, we shall see how a collaborative knowledge management system (CMKS) can be built using the JPR agent library. We assume that each responder/volunteer has an Android device. All the personnel involved in response activities report to a disaster control center equipped with a machine running Java.

3.3 JADE - A FIPA-compliant agent framework

In the initial stage of this project, we surveyed the literature pertaining to agent-systems with the aim of finding a suitable framework to linkup JPR with. The overview of agent-systems presented in section 2.2 is an outcome of this survey. Extending Gasser's views [2001], Sturm and Shehory [2014] proposed the following set of general requirements that a multi-agent framework is expected to meet:

1. Heterogeneity: A framework should be able to support different kinds of architectures and allow interaction with various external entities devoid of their nature (agent or non-agent).

2. Scalability: It must be highly scalable.

3. Standardization: It should adhere to some standard specifications.

4. Support: It must evolve continuously to support modern applications.

We chose the Java Agent Development framework (JADE) [Bellifemine et al., 2007] for this project because it fulfills all the above requirements.

1. JADE does not enforce any type of agent architecture. Through HTTP and other standard networking protocols, it supports interaction with non-JADE agents.

2. It was fully written in Java, as is JPR (excluding some support libraries), thus favoring an easy integration of the two. Moreover, it provides special support libraries for Android.

3. It is fully FIPA-compliant.

4. JADE is open source.

Following five years of initial development by Telecom Italia, JADE went open source in 2000. Companies like Motorola, Siemens etc. and several universities across the world are actively

involved in its development [Bellifemine et al., 2007]. Therefore, it has been widely used in the past, both for academic research and industry-level multi-agent applications [Bogdanova et al., 2014; Su and Wu, 2011; Zhao et al., 2007; Bădică et al., 2006].

3.3.1 Architecture

JADE agents are in accordance with Jennings and Woolridge's [1995] definition of an agent discussed in the previous chapter. It facilitates the development of software agents as distributed code processes that can communicate with each other. The building blocks of the JADE framework are the*agent platform, containers* and the *agents*.

The Agent Platform (AP):

An agent platform is the top most element in the FIPA abstract architecture specification. It is composed of machines, operating systems, and agent support software. It is responsible for providing the physical infrastructure for agents to live and execute their actions. FIPA also specifies the services that an agent platform must provide, including agent management, yellow pages etc. Based on this abstract element, all the communications in the system are either inter-platform or intra-platform and specifications are provided for each. The actual implementation of the platform is left to the developer. In JADE, a platform is implemented as a set of connected *containers*. An instance of JADE agent platform is shown in Fig. 3.1.

JADE does not impose any restriction on the architecture of a multi-agent system. Also, the developer is free to choose a reactive, deliberative or a hybrid model for the internal architecture of an agent.

In our motivating scenario, the central computer (a PC or a server) hosted at the disaster control center and the Android devices that the responders carry provide the physical infrastructure for launching an agent platform.

Containers:

A *container* is an instance of the JADE runtime environment. JADE implements the agent platform as a set of containers, with a special container providing the services mandated by FIPA. Agent containers are not a part of the FIPA abstract architecture specification. Instead, they are the software elements in the JADE framework that constitute an agent platform. They provide a runtime environment for agents which are implemented as Java threads. They can be deployed on machines running Java or Android OS. Launching an agent platform in JADE is equivalent to creating a special type of container called the 'main-container', which hosts two special agents - AMS and DF, and maintains the addresses of all the other containers created thereafter. The directory facilitator agent (DF) provides yellow pages services. The Agent Management System (AMS) agent maintains the IDs of all the agents in the platform.

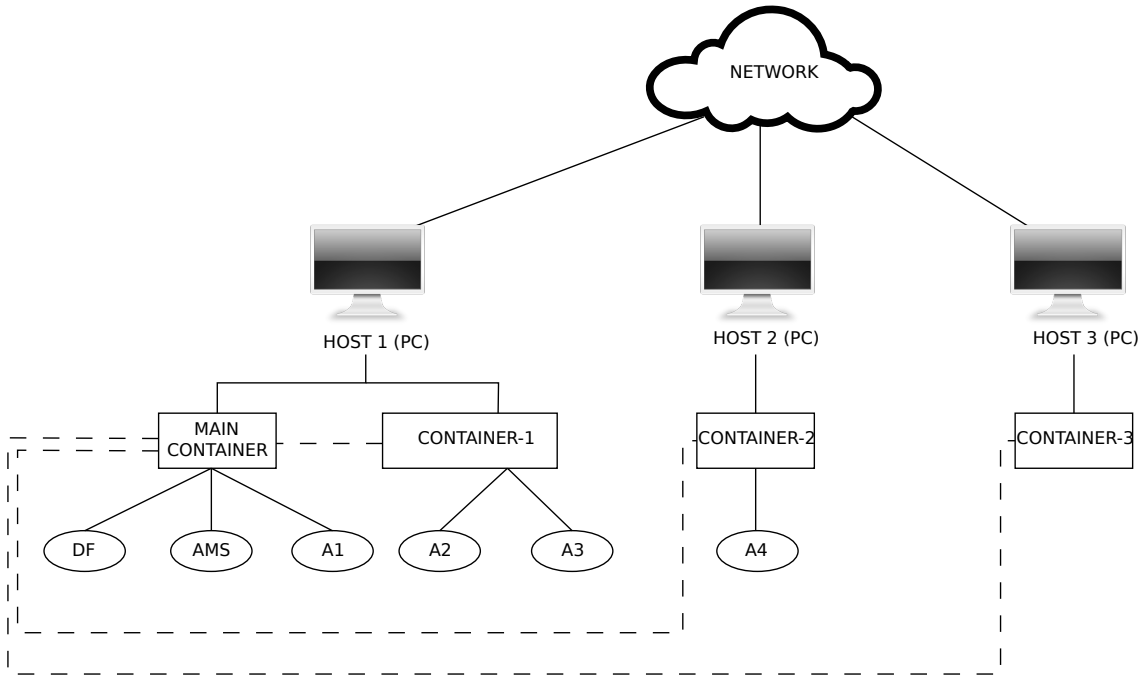

Figure 3.1: A JADE platform running on three different machines

Work on JADE began around 1995, and so it was not initially planned for mobile devices. Some of the design decisions made for containers result in the following short-comings when implemented on mobile devices:

- The platform management services provided by the main-container are essential for the functioning of a peripheral container. JADE assumes a constant connection between the two. This cannot be guaranteed in a mobile environment.
- JADE cannot efficiently handle the dynamic IP address allocation and high network latencies often encountered in mobile networks.
- Containers have a high memory footprint. This is problematic for resource-constrained (mobile) devices.

Therefore, for supporting distributed applications on mobile devices, JADE offers a separate execution mode for containers called the 'split-execution mode'. To differentiate them from the normal containers we have been discussing so far, we call them *split-containers*. In this mode, a normal container is *split* into a front end hosted on the mobile device and a back-end hosted on the same machine as the main-container (that is, the central server). The tasks assuming constant connection with the main-container are executed on the back end. The front-end and back-end communicate over a possibly intermittent wireless connection. As indicated in table 3.1, the front-end is much lighter compared to a normal-container.

Table 3.1: Division of responsibilities between the front-end and the back-end

Full Agent Container	Back End	Front End
Platform Service Management	Platform Service Management	Agent Management
Platform Service Finding	Platform Service Finding	Comm. with back-end
Comm. with main-container	Comm. with main-container	
Agent Message Handling	Agent Message Handling	
Agent Management	Comm. with front-end	

Given the different modes of execution for containers, some suitable architectures for modeling the disaster scenario are:

Configuration-I: Because a disaster control center (DCC) is the central entity of response activities, one obvious way to model the disaster response scenario is to create a main-container on the hardware at the DCC (say DCC-PC). All the other (Android) devices can launch peripheral (non-main) containers. TCP/IP connections between the main and peripheral containers (in other words, the DCC-PC and Android devices) is assumed. This configuration is shown in figure 3.2.

Configuration-II: Like in the previous case, the main container is created on DCC-PC. Split-containers can be launched on Android devices, carried by the volunteers and responders. This configuration is more robust 1) due to a lighter front-end and 2) reduced dependency on the wireless connection. This configuration is shown in figure 3.3.

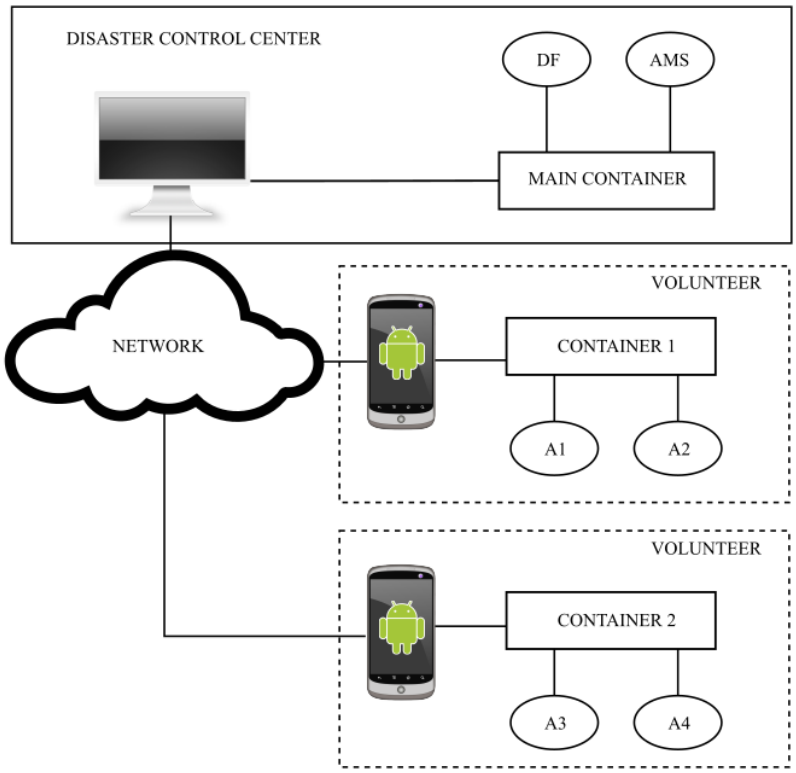

Figure 3.2: Configuration-I: Full containers on Android devices

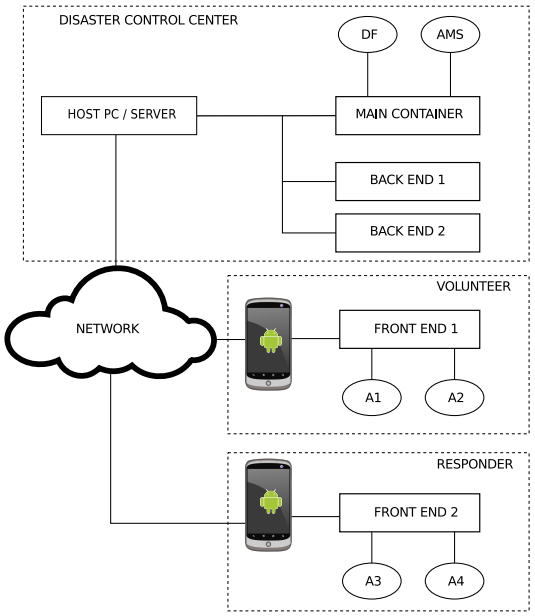

Figure 3.3: Configuration-II: Split-containers on Android devices

Configuration-III: In the very initial stages of response it is possible that a DCC has not been set up yet. Response activities are carried out by groups of volunteers, each led by an emergency responder. In this case, an alternate configuration where an emergency responder hosts a main-container, and the volunteers in his/her group host peripheral containers is possible. All the tasks performed by the DCC-PC can be performed reasonably on a emergency responder's device as well, although on a much smaller scale.

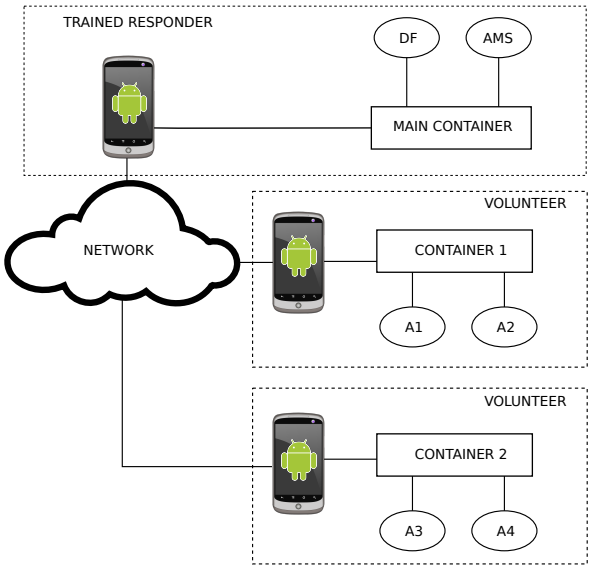

Figure 3.4: Configuration-III: Ad-hoc network setup

In the rest of this chapter, we focus on configuration-II, which is more suitable for real-world scenarios. The availability of a resource-abundant machine (DCC-PC) to reason over huge amounts of data collected from various devices is a particular advantage. Nonetheless, using JAL it is possible to realize any type of configuration discussed so far. The only restriction, imposed by JADE and consequently JAL, is that a main-container cannot be launched in split-execution mode. This implies that a machine with a reasonable computing power should host a (complete) main-container. Chmiel et Al. [2005] performed a series of experiments with thousands of JADE agents on machines with RAM as a low as 48-192 Megabytes . Since most of the mobile devices available today are much more powerful (usually 1 gigabytes of RAM or more), this is not a serious limitation.

Agents

Agents are implemented as threads running in the context of their respective containers (processes). Each agent is identified by its Agent ID (AID) which consists of

- a local name specified by the user during its creation;

- a full name, concatenation of local name, "@", and container name;
- and the address(es) of the container it resides in.

No two agents in a platform can have the same local name. *Behaviors* are the tasks to be performed by an agent. Following are the basic types of behaviors defined (built-in) JADE:

- One shot behavior - Performs a given task (a piece of Java code) once
- Cyclic behavior - A task or set of tasks that are performed continuously in a cyclic fashion
- Generic behavior - Combines different types of behaviors into one. The appropriate behavior is triggered based on a switch.
- Waker behavior - The execution of which starts after a given timeout.
- Ticker behavior - The execution of which occurs at regular intervals of time.

One or more behaviors can be added to an agent's *behavior queue*. One shot behaviors are executed and discarded, where as cyclic behaviors are executed and placed at the end of the queue for their next turn. Behaviors can neither take precedence in order of execution nor interrupt other behaviors. Therefore, the developers should carefully program the behaviors so that they end promptly. JADE also provides mechanisms for running behaviors on dedicated threads. However, it does not provide much support for synchronization. More complex behaviors can be programmed using other subclasses of the 'Behaviour' class such as Finite State Machine (FSM) behaviors, sequential, serial and parallel behaviors. A detailed description of these behaviors is provided in the JADE manual [Bellifemine et al., 2001].

In the disaster response scenario, we assume that each container in the system hosts only one agent. In our discussion of the agent library, we provide concrete examples of the behaviors of the agents on the DCC end and the volunteer/responder end. Agents communicate by sending FIPA-ACL messages. The content of a message can be any serializable Java object.

3.3.2 Access from External Applications

External Java applications can use the following JADE classes to create agent containers and thus agents:

- `jade.core.Runtime` to create full containers
- `jade.core.MicroRuntime` to create split containers

To be consistent with Android's philosophy of implementing long-running activities as services, JADE provides implementations of the above classes as services, namely the `jade.android.RuntimeService` and `jade.android.MicroRuntimeService`. JADE agents, to preserve autonomy, do not provide callbacks or allow access to their object references from external applications. Instead, when an agent or a container is created, the runtime instance returns wrapper objects (container controller and agent controller). The interaction with a container/agent should proceed through their respective wrapper objects.

Alternatively, for interacting with an agent, developers can create a custom interface and have the agent expose it to external applications. This scheme internally uses Java's dynamic proxy mechanism [2], to preserve agents' autonomy.

3.4 JPR

Having provided a basic overview of the JADE framework, in this section, we provide a similar introduction to JPR. We assume that the reader has an understanding of the basic concepts of Prolog such as terms, backtracking, resolution etc. We rather focus on the implementation details including organization of key Java classes and their purpose. The `jpr.engine.InferenceEngine` is the key component that developers using JPR would be dealing with.

The `jpr.io` package contains a set of classes which manage the input and output streams for the inference engine. Each instance of the inference engine has attached to

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

it an `jpr.io.IOController` which can be used to set or modify its IO streams. The `jpr.io.PrologReader` uses the inference engine to parse a string or a file as Prolog terms.

A general interface called `jpr.terms.PrologTerm` and the classes implementing it provide definitions for variables, numbers, structures, and additionally *reference terms* in Prolog. Reference terms are not a part of the ISO standards for Prolog. They are used to wrap arbitrary Java objects, allowing them to be used in Prolog programs.s Structures are compound terms, with a functor and a set of Prolog terms as arguments. Prolog 'atoms' are structures with zero arity.

JPR implements the factory method pattern to create instances of these terms. The `jpr.terms.TermFactory` interface extends the respective factory interfaces for numbers, variables and reference terms. A default implementation of the term factory, attached to the IE, is used to create various Prolog terms. Structures are created using structure factories defined for each predicate. Each structure is defined in a class with an `execute` method which is used by the inference engine at run-time. Specifically, the execute method controls what happens when a structure is called as a goal. A map of predicate keys(name of the predicate, arity) and the corresponding structure factories is maintained; this controls which predicates are created when a Prolog file is parsed.

Each library (Built-in or custom) in JPR defines structure factories for predicates. As mentioned earlier, each predicate is typically defined in a class of its own. The libraries implement the `jpr.libraries.PrologLibrary` interface, which mandates the `keys` and `factories` methods. The JPR agent library (JAL) defines a set of predicates for creating and managing agent systems directly from Prolog. The predicates are internally implemented in Java and use the JADE API.

Each instance of the inference engine class has attached to it a knowledge base which acts as a storage for rules, a list maintaining variable bindings at runtime, and a stack to store the backtrack points that are encountered during the execution of a program. The `solve` method of the `jpr.engine.InferenceEngine` class, which takes strings or Prolog terms as

inputs is responsible for solving the queries. Files can be loaded into a knowledge base using a consult method on the respective inference engine. For a simple application using Prolog in the background, all that the developer needs to do is:

1. Create an instance of the `jpr.engine.InferenceEngine`(IE) and set its IO streams.
2. Add rules to the engine's knowledge base using its `consult` method.
3. Use the `solve` method to respond to application-related events such as button clicks etc.

3.5 JPR Agent Library (JAL)

The aim of the JPR agent library (JAL) is to interface Java Prolog (JPR) and JADE, thus facilitating the creation and maintenance of agents through JPR. We discuss the design of the agent library, implementation of various predicates, and its limitations. JADE is a rather sophisticated framework, providing numerous functionalities. As a first step towards the integration of JPR and JADE, we limit ourselves to providing hooks to a small subset of these functionalities capable of supporting a context-aware multi-agent system.

Organization of the library

The library is organized in to three packages: `agentIE, deviceIE, utils`. The utils package is composed of auxiliary Java classes which contains methods that several predicates in the library commonly use.

In the following discussion, we shall see that each agent in the system bears its own inference engine. The agentIE package contains Java classes that define these predicates. But additional predicates are required to create containers and thus agents in the first place. The classes defining these predicates are grouped under the deviceIE package. In the following section, the difference between the two will be made more clear to the reader.

The `DeviceLibrary.class` in the deviceIE package is the bootstrap point for the library. Two separate constructors are provided for this class, one for loading the library on a PC and another for Android devices. Each instance of DeviceLibrary.class has attached to it, a AgentLibraryManager.class instance. The *manager* stores the object references of JADE runtime instances, service connections (on Android), and other key elements. The manager instance is passed to all the predicates in the library.

3.5.1 Implementation

In this section, we discuss the implementation of various predicates in JAL and demonstrate their usage through simple examples. The examples will be based on the disaster response scenario. As mentioned earlier, we implement the configuration-II described in section 3.3.1. We assume that the following physical infrastructure for hosting a multi-agent system (JADE agent platform) is available:

- At the disaster control center, a PC with the latest version of Java installed.
- An Android device for each volunteer/responder participating in disaster response.

On the DCC-PC, lets assume that a Java application is created and is running an instance of JPR (jpr.engine.InferenceEngine). A main-container can be created using the `mainContainer_create/4` predicate. This predicate takes the IP address at which the main-container is to be created, port, name, and a variable as arguments. In the DCC case, the IP address can be same as that of the DCC-PC. This returns a reference to the created Java object of `jade.wrapper.ContainerController.class` (wrapper for a container).

```
% On DCC-PC
?- mainContainer_create('192.168.2.14', 1099, dccMain, X).
X = @dccMain
...
```

To make a container accessible from Prolog, the controller object is wrapped in a Prolog reference term with the name of the container(atom) as its handle. This reference term is then stored in a Java HashMap, maintained by the agent library manager, with its handle as the key. In the rest of the discussion we refer to this map as the 'container controller map'. The variable `X` passed in the above example is bound to the newly created reference term. Multiple main-containers can also be launched from a single instance of JPR. But for the sake of simplicity we limit our discussion to a single main-container. Peripheral containers can be created in a similar fashion using the `container_create/4` predicate. In this case, the host and port arguments correspond to that the main-container.

After a container is created, its reference term can be used to manage it from the Prolog side. For example, passing a reference term, wrapping a container controller, to the `container_kill/1` predicate terminates the corresponding container. To perform this operation at a later stage when the reference term is not available, developers can pass the name of the container as the argument.

Moreover, when a container is created, a fact is asserted to the knowledge base with `container` or `mainContainer` as the functor and the name of the agent (atom) as the argument. Using these assertions, the developer can retrieve a list of peripheral containers from the KB using `forall/3`, `setof/3` or such predicates, and perform desired operations on each. At any point in the program, the developer can get the reference term from an atom using the `getRefObject/1` predicate. If multiple operations are to be performed on the same container, it is recommended that the developer first obtain the reference term from the atom using this predicate and pas7s it to the consequent goals. This would enhance the performance because now the program does not perform the same look up multiple times.

On an Android device, JADE runs as a service. Currently, JADE does not support multiple containers on an Android device. However, there is no restriction on the type of a container (split, full or main) or number of agents. To use JPR in an Android application, an instance of `jpr.engine.InferenceEngine` must be created in one of the activi-

ties[3]. To load the agent library, the developer must use an additional constructor provided for the `jal.main.AgentLibrary` class which takes the `android.content.Context` of an application and a boolean `isSplit` value as parameters. The latter indicating if the developer wants to launch a split-container or a stand-alone container (full or main). The corresponding service i.e. `jade.android.RuntimeService` for a stand-alone container and the `jade.android.MicroRuntimeService` for a split-container, must be declared in the manifest file of the application as shown below.

```
...
<application
   android:allowBackup="true"
   android:icon="@drawable/ic_launcher"
   ...
   <service android:name="jade.android.MicroRuntimeService" />
      <activity
         android:name=".MainActivity"
         android:screenOrientation="portrait"
         ...
```

An agent library manager is created passing the values of context and *isSplit* parameters to its constructor. The manager, in its constructor method, attempts to create a `android.content.ServiceConnection` between the context (activity) and the service declared in the manifest file. This occurs in a thread separate from the one creating the manager object. If a service connection is established successfully it returns a binder object of type `jade.android.RuntimeServiceBinder` or `jade.android.MicroRuntimeServiceBinder`. The reference to the binder object is stored by the agent library manger for various predicates to use it.

To create containers on Android devices, the following set of predicates are defined.

[3]We assume a basic understanding of the Android frameworkReaders lacking this may refer: https://developer.android.com/guide/index.html

- `fullMainContainer_create/0` can be used to create a main-container. This automatically retrieves the IP address of the Android device and creates the main-container at the same address.

- `splitContainer_create/3` can be used to create a split-container. The IP address, port of the main-container and the name of the container to be created should be passed as arguments

- `container_create/3` can be used to create a full-container. This predicate also takes the IP address, port of the main-container and the container name as arguments

In Android, any interaction between an android activity and a service must proceed in an asynchronous fashion. So, when a container is created, the JADE framework returns the wrapper objects using a callback[4] mechanism. When a callback is received, a reference term wrapping the returned object is created and stored as a fact in the knowledge base (using `(container/1)` predicate).

On a PC, agents can be launched using the `agent_create/4` predicate. This predicate takes the name of the agent, name of the container or a reference term wrapping it, the Java class of the agent, and a variable as arguments. A default agent class is provided (`jal.deviceIE.agents.DefaultAgent.class`), which can be passed as the third argument to the `agent_create/4` predicate. Using the second argument as the key, this predicate retrieves the corresponding container controller object from the *container controller map*. It calls the `createAgent` method on the controller object. This call returns a `jade.wrapper.AgentController` object that wraps the newly created agent. This object is in turn wrapped in a reference term and stored in a map maintained by the agent library manger.

On an Android device, agents can be created using the `agent_create/2` predicate which takes the name of an agent and the Java class. Internally, this predicate triggers the

[4]http://docs.oracle.com/javase/7/docs/api/javax/security/auth/callback/Callback.html

`createAgent` method on a container wrapper object. An empty callback is received after an agent is created. On receiving this callback, a fact is asserted to the knowledge base with `agent` as the functor and name of the agent as the argument.

Coming back to the motivating scenario, the code snippets for creating containers and agents is shown below. Let us consider volunteers Bob and Alice and name the agents on their devices accordingly.

```
% On Bob's device (Android)
?- splitContainer_create('192.168.2.14', 1099, container1).
...
?- agent_create(bob,'jal.deviceIE.agent.DefaultAgent.class').
```

```
% On Alice's device (Android)
?- splitContainer_create('192.168.2.14', 1099, container2).
...
?- agent_create(alice,'jal.deviceIE.agent.DefaultAgent.class').
```

Agent Behaviors

A default Java agent class called the `jal.deviceIE.agents.DefaultAgent.class` which extends the `jade.core.Agent` class is bundled with the library. Developers can use this default class unless some special Java libraries for JADE agents, are desired on the Java side. JAL defines a set of low level behaviors in Java. Predicates to initiate and manage these low-level behaviors are provided on the Prolog side. The rationale is that the developers can program more complex behaviors, in Prolog, using these low-level predicates. Since agents are the work horses of the system, we provide a detailed description of the default agent class, the behaviors of an agent, communication and respective predicates.

An instance of the `jal.deviceIE.agents.DefaultAgent.class` has attached to it an inference engine, a list to maintain variable bindings at runtime, and an instance of

`jpr.io.PrologOutput` that wraps a (byte array) output stream. Each time an agent is created using the `agent_create/4` predicate, its `setup` method is called. The setup method, initializes the inference engine, and the array list objects. It also creates a reference term, wrapping the controller of the agent object and stores it in the agent's knowledge base as a `myAgent(controllerRefTerm)` fact. It is important to note that the agent's inference engine, and therefore its knowledge base is different from the device inference engine i.e. the engine which is executing the `agent_create/4` predicate. This is illustrated in fig. 3.5 where a container with three agents is launched from an Android device.

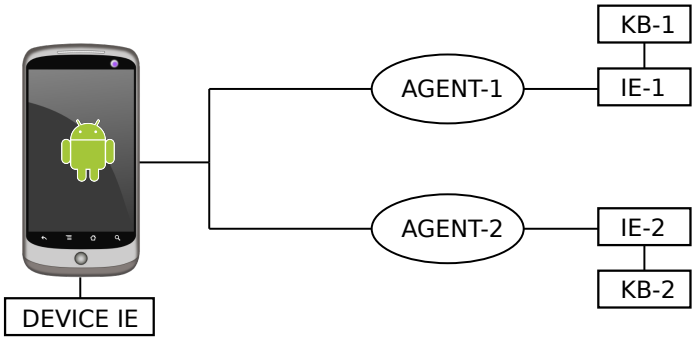

Figure 3.5: Two agents hosted on an Android device, all three having distinct inference engines

Therefore each agent with its own inference engine and knowledge base acts as an autonomous reasoning entity whose behaviors can be triggered from the device level IE at run time. The suggested way of implementing agent behaviors is by creating nested classes in the agent class. In JAL, the `jal.deviceIE.agents.DefaultAgent.class` has nested behavior classes implemented for one shot, cyclic, ticker and waker behaviors discussed in sec. 3.3.1. Each behavior class has a constructor which takes a query string as a mandatory argument and any additional arguments. For example, the ticker behavior takes the query string and time interval as arguments. When a behavior is put to execution, the agent takes the string argument and solves it using its inference engine. With the idea roughly stated, we demonstrate the whole process using an example.

Consider the disaster management scenario described in Section 3.2. Suppose, for real-time monitoring purposes, the control center wants to know the location of the emergency responders at regular intervals of time, say every 60 seconds. Here we describe how this can be achieved using the *dccAgent* and the agent named *bob*. It is assumed that the agents hosted on the Android devices able to access the sensors of the device it is hosted on. For this purpose, we use the Android sensor library for Prolog developed at UGA. The sensor library can be loaded to the agent's inference engine using the `agent_loadSensorLib` predicate.

A behavior file defining the behaviors of agent *bob* (on the volunteer end) as Prolog rules must be created. From the device level inference engine, this file can be loaded using the `behaviorFile_load/2` predicate, that takes the name of the agent and file path as arguments. File IO on Android is slightly different from that on Java. The behavior files should be placed in the assets folder of an Android application project for the agent to be able to access it. The predicate internally gets the agent controller object, from the name of the agent, gets the O2A interface, and calls the DefaultAgent's Java method *loadBehaviorFile* passing the file path as a parameter. This method uses the consult method of the inference engine class to load this file to its knowledge base. A sample behavior file is shown below. The `setSom` rule creates a sensor observations manager, and sets the window size to 1. The `lastObservation/2` predicate returns the most recent observation. The `message_inform/2` predicate sends a message to another agent. We will discuss the inter-agent message passing in the following section.

```
%bobBehav.pl
setSom:- createSom(gpsSom,gps,fastest),
       setCapacity(gpsSom,size,1).

sendGPSData:-   lastObservation(gpsSom,GPSValues),
           getValues(GPSValues,[Lat, Long]),
           message_inform(dccAgent,[Lat, Long]).
```

Once the behavior file is loaded, 'setSom' is added as a one shot behavior. The `tickerBehavior_add/3` predicate is used with the first argument as the name of the agent, the second argument as the head of the rule describing the behavior in bob's behavior file. In our case, the 'sendGPSData' rule and the third argument as the number of seconds it has to wait after each execution. Internally, this predicate calls the O2A interface on the agent controller object, and calls the DefaultAgent's Java method *addTickerBehavior*, passing the last two argument as parameters. This Java method creates an instance of the `TickerBehaviorQuery` class, passing the same two arguments to its constructor. This object is added to the behavior queue of the sensor agent and the `tickerBehavior_add/3` predicate returns. Now the agent executes the 'sendGPSData' action every 60 seconds without the intervention of the user.

```
% On Bob's device (Android)
?- spliContainer_create('192.168.2.14', 1099, container1).
...
?- agent_create(bob,'jal.deviceIE.agent.DefaultAgent.class').
...
?- X = bob, getObject(X,Y),
   behaviorFile_load(Y, 'bobBehav.pl'),
   oneShotBehavior_add(Y, setSom),
   tickerBehavior_add(Y, sendGPSData, 60).
```

A sub-library is also provided which defines predicates for behavior loading and message passing directly from the agent's behavior file. For example, the `behavior_add/2` predicate adds a one shot behavior, performing some trivial operation each time the 'sendGPSData' predicate is triggered. The 'trivialOperation' is added to the end of the agent behavior queue and is executed when all the preceding possibly more important behaviors in the queue are executed. The `behavior_add/2` predicate can also be used to create nested behaviors on the Prolog side. Internally, this predicate and all the other predicates in the sub-library uses the *myAgent(X)* assertion, made at agent start-up to get a handle on its controller object. The corresponding methods for each predicate are triggered via the O2A interface. The default

agent class, in its setup method, loads this sub-library to the agent inference engine. So it is completely transparent to the developer. A rule of thumb: all the predicates in JAL that take agent name as an argument are executed primarily by the deviceIE and the ones that do not are executed only by the corresponding agent‘s inference engine. The message passing predicates defined in the further sections also belong to this sub-library.

```
%SensorDataAggregator.pl
:- setSom, behavior_add([ticker,60]sendGPSData).
sendGPSData:- lastObservation(gpsSom,GPSValues),
              getValues(GPSValues,[Lat, Long]),
              message_inform(dccAgent,[Lat, Long]),
              behavior_add(1, trivialOperation).

setSom:- createSom(gpsSom,gps,fastest),
         setCapacity(gpsSom,size,1).

trivialOperation:- ..
```

As mentioned earlier, JADE offers more complex behaviors such as Finite State Machine behaviors, Sequential behaviors etc. These behaviors build upon the simple JADE behavior classes we discusses so far. JAL provides hooks to these basic classes. Therefore complex behaviors of an agent can be programmed on the Prolog side leveraging the basic JAL predicates. Moreover, behaviors can be modified dynamically at runtime by editing and re-consulting the behavior files of an agent.

Agent Queries

An agent created via JPR acts a separate reasoning entity with its own knowledge base. It can be queried from the device inference engine using the `agent_query/3` predicate. This predicate takes the name of an agent, the query (enclosed in quotes), and a list of variables as arguments. The query is placed to the corresponding agent via the O2A interface. The agent solves the query and returns a list of variable bindings.

The variables returned by the agent and the variables in the input list (third argument) are matched lexically. Each input variables is unified with a copy of the term the lexically matching variable in the list of agent bindings is bound to. In this way, multiple agents can be queried at the same time. The functioning of the `agent_query` is more complex than one might usually expect because it essentially involves two different inference engines exchanging Prolog terms.

Agent Communication

The inter-agent communication model in JAL is shown in Fig. 3.6. An agent can send a message to another agent using the `msg_inform/2` predicate which takes the name of the receiving agent and the message to be passed as arguments. Note that this predicate is used in the behavior file of an agent and not the device inference engine. If due to some reason, the message sending behavior has to be triggered from the device inference engine, the developer can define a rule in the agent behavior file that has the `msg_inform/2` predicate in its body.

Internally, this predicate uses the `toString` method on the Prolog term passed as the second argument. It creates an `jade.lang.acl.ACLMessage` object with its performative as `inform` and its content as the string returned by the `toString` method. The message is sent using the `send` method of JADE's `ACLMessage` object. The message passing mechanism is asynchronous. So the `msg_inform/2` returns immediately after sending the message. The receiving agent wraps the message in a reference term and triggers the `onMsgReceive(+Variable)` rule, passing the reference term as the argument. The developers must reserve the `onMsgReceive(+Variable)` rule for this purpose and define how an agent must respond to incoming messages. JAL provides `msg_getPerformative/2` and `msg_getSender/2` predicates to obtafin the performative and sender of a message. To get the content of the message as Prolog terms, the `msg_getTerms/2` predicate can be used. It internally uses the inference engine of the receiving agent to parse the content string as Prolog terms. Therefore what happens when a message is received is fully in the hands of

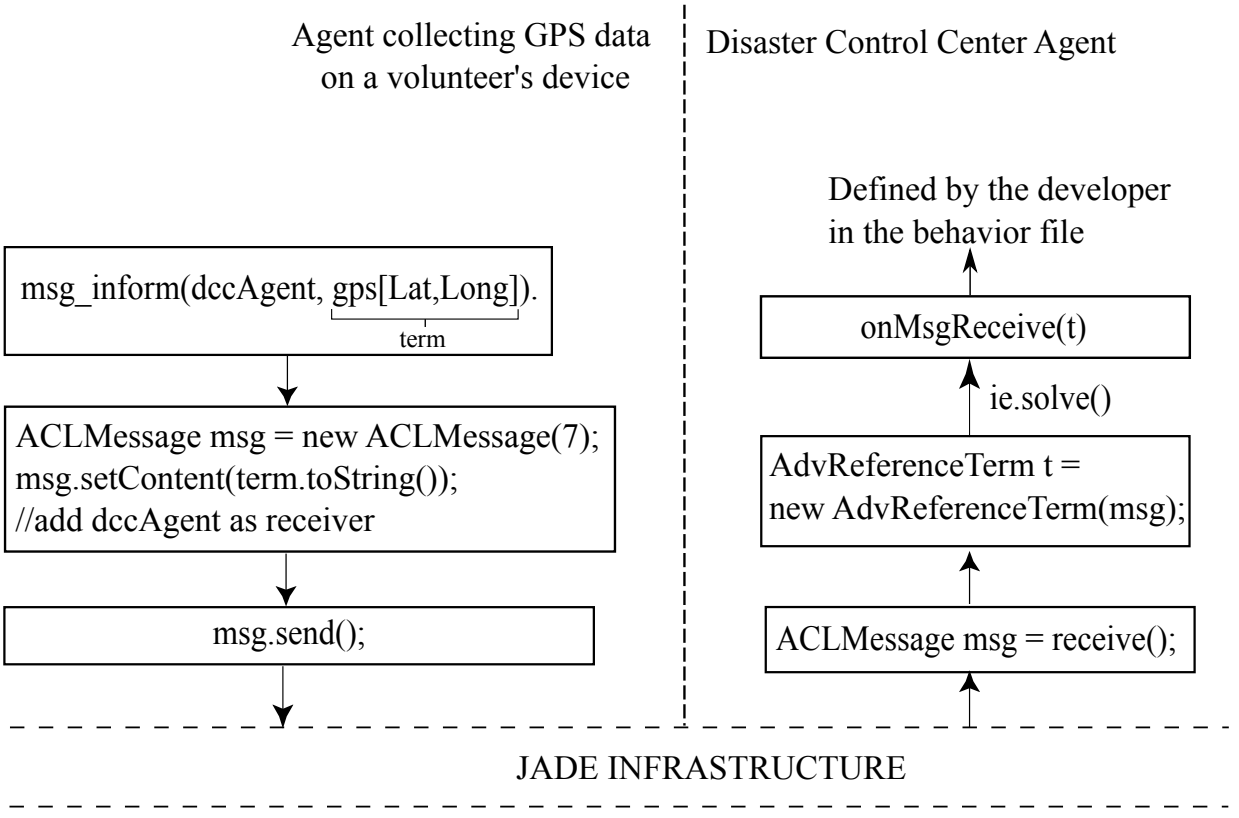

Figure 3.6: Inter-agent communication in JADE

the developer. For example, the agent hosted at the disaster control center (dccAgent) to collect the location data of various volunteers agents is shown below.

```
%dccAgentBehavior.pl
onMsgReceive(X):-  msg_sender(X, NameOfVolunteer),
                   msg_performative(X, Performative),
                   msg_contentAsTerm(X, Term),
                   act(Performative, NameOfVolunteer,Term).

act(inform, Volunteer, gps([Lat,Long])):-
assert(location(Volunteer,Lat,Long)).

act(inform, Volunteer, victimData(Name, Age, Sex, Condition)):-
%To handle victim data
...
```

3.6 Conclusion and Future Work

In this chapter, we discussed the design and implementation of the agent library (JAL). We have described the method of creating agents, configuring their behaviors, querying and message-passing. In Appendix-A, we discuss the disaster management application at length including a couple of use cases. Since JADE is a huge framework, we have limited ourselves to implementing the features mandated by FIPA. The following is a list of some additional features in JADE that JAL does not provide hooks to:

- support the use of message content ontologies.
- provide enough support for debugging the agent system (does not provide hooks to the debugger agents in JADE)
- implement various interaction protocols to hold complex conversations
- provide means to use the DF and AMS agents effectively (no corresponding predicates)

But even in its current form, JAL is capable of supporting quite a good number of applications. It is in par with most of the existing frameworks for declarative agents. More importantly, the JAL in its present state is good enough to support a context-aware multi agent system. This is the prime focus of the following chapter.

Chapter 4

Computing the Well-Founded Model of a Multi Context Systems

4.1 Introduction

This chapter follows from the previous which presents an agent library for JPR. The library was used to implement a distributed knowledge management system (DKMS) where each agent, with its own knowledge base and inference engine, acts as an autonomous reasoning entity. Knowledge sharing is facilitated through a simple message passing scheme. Based on the experiments performed by Chmiel et Al. [2005] for testing the performance of JADE, we have concluded that this scheme is computationally viable.

However, this system of distributed reasoning agents does not meet the requirements of a distributed knowledge management system. Specifically,

- Each agent KB contains a normal logic program written in Prolog. Without a proper semantics (such as well-founded [Van Gelder et al., 1991] or stable-model [Gelfond and Lifschitz, 1988]), the interpretation of programs involving default negation (*not*) is problematic. In section 2.4.1, we have discussed this problem in more detail.

- The knowledge dependencies between different nodes cannot be declaratively specified. For example, consider two agents, with respective knowledge bases KB_1 and KB_2, reasoning about an entity. Rules such as "if p holds in KB_1, then q holds in KB_2" are not specified. If these rules are accommodated, the flow of information, from one agent to another, can be entirely managed by the framework (making it transparent to the developer).

- In many cases a global view/model of the distributed system is desired. But JAL does not provide a way of computing a global model. In context-aware systems a global model encompassing all the agent KBs provide a complete picture of the entity being reasoned about.

In this chapter, we set out to resolve these issues by providing an enhanced reasoning scheme for distributed knowledge management systems. It is based on a stream of research in logic-based AI which attempts to formalize the notion of context/contextual reasoning [Guha, 1991; McCarthy, 1993; Giunchiglia and Serafini, 1994; Brewka et al., 2011; Antoniou et al., 2010]. Of particular interest here is the non-monotonic multi-context systems (MCS) framework proposed by Brewka et Al [2007; 2007; 2011; 2013]. It provides *bridge rules*, as a way of formally representing interrelations between different *contexts* (roughly equivalent to KBs) and, more importantly, a *global semantics*. There is abundant theoretical work (starting with McCarthy's [1989]) in this area but only a handful of implementations (including Moawad et al. [2013]; Dao-Tran [2014]. Even in these few implementations, there is little discussion on the applicability for mobile platforms. This is the primary motivation for the work presented here.

4.2 Using a Multi-Agent System to Implement a MCS

A distributed knowledge management system contains several autonomous computational entities which facilitate knowledge storage and retrieval. The autonomous computational entities in our implementation are the *agents* launched through JPR. To leverage the non-monotonic multi-context systems framework [Brewka et al., 2011], each agent KB is treated as a *context*. In this section, we briefly reiterate some concepts of the MCS framework and explain how a global model is computed.

Definition 4.2.1. *[Brewka et al., 2011] A multi-context system $M = \{C_1, C_2, ...C_n,\}$, is a collection of contexts, where each context is a tuple*

$$C_i = (L_i, kb_i, br_i), \ where$$

$$L_i = (KB_i, BS_i, ACC_i) \ is \ a \ logic,$$

kb_i is a knowledge base (an element of KB_i), and br_i is a set of L_i-bridge rules over L_1,..L_n. BS_i and ACC_i are the belief sets and acceptable belief sets respectively.

We take an agent-based approach for implementing a multi-context system. Each agent launched through JAL has its own knowledge base and inference engine. To represent the bridge rules in an agent KB, we provide the predicate `br/2` which takes the name of an agent(context) as the first argument and a Prolog term as the second. This representation allows us to leverage inter-agent communication to check if a bridge rule is satisfied (or not). Below, we decide the logic L_i for each agent KB.

The stable model/answer set semantics [Gelfond and Lifschitz, 1988, 1991] and the well-founded semantics [Van Gelder et al., 1991] are the two dominant semantics available for normal logic programs. Using an iterative procedure, the well-founded model can be computed in polynomial time with respect to the size of the program. On the other hand, it is NP-hard to decide if a program has a unique stable model. Since we are working with resource-constrained (mobile) devices, complexity is a crucial factor. So each logic L_i in our implementation is a normal logic program under the well-founded semantics. Consequently, any set of literals belonging to KB_i is a belief set, and the acceptable belief set ACC_i is the well-founded model. By adhering to the same L_i in all the contexts, we are working with a homogeneous system.

The semantics of a MCS is defined in terms of *belief states*, where a belief state is a sequence $S = \{S_1, S_2, ...S_n\}$ such that each S_i is an element of BS_i. Intuitively, a belief state is a combination of belief sets, one from each context.

Definition 4.2.2. *[Brewka et al., 2011] A belief state $S = (S_1, ..., S_n)$ of a multi-context system M is an equilibrium iff $S_i \in ACC_i(kb_i\ U\ (head(r)|r \in app(br_i, S)), 1 \leq i \leq n.$, where $app(br_i, S)$ means applicable bridge rules in i, given the belief state S(belief sets of all other contexts).*

Intuitively, a belief state is an equilibrium if each of the component belief sets are acceptable, given others. Brewka and Eiter [Brewka and Eiter, 2007], theoretically, prove that the equilibrium of a MCS under well-founded semantics (WFS) can be computed in polynomial time (with respect to size of the program). The rest of this chapter is devoted to explaining how the equilibrium (a global well-founded model) can actually be computed using our implementation of a MCS.

We extend the JPR agent library, by modifying the behavior of the *default agent* on the Java side, and introducing a special type of agent called a *coordinator agent* (`CoordinatorAgent.class`) which will coordinate the computations carried out by each agent. The extra set of predicates that are introduced in this chapter are bundled into the JPR Context Library (JCL).

4.3 Computing a Global Well-Founded Model

In this section, we examine the contextual dependencies i.e. how different contexts in a MCS are connected through bridge rules, in more detail. An algorithm for computing a global well-founded model is presented and its implementation is discussed.

Example 4.3.1. *Let us consider the following MCS, which we use as a running example through the rest of this chapter:*

$$
\begin{aligned}
&M = \{C_1, C_2, C_3, C_4\} \\
&C_1 = \{b.,\ a :- b.)\} \\
&C_2 = \{p :- br(a1, b).,\ q :- not(br(a3, r)).\} \\
&C_3 = \{r :- not(br(a2, q)).\} \\
&C_4 = \{w :- br(a3, r).,\ x :- w.\}
\end{aligned}
$$

We use the predicate *br/2* to indicate bridge terms. This takes the subscript of a context as the first argument and a Prolog term as the second. Intuitively, the first rule in C_1 means that: if b holds in C_1, then p holds in C_2. Therefore, a system implementing this MCS must facilitate the flow of information from C_1 to C_2, about the truth/falsity of p. We call a directed graph representing the flow of such information in a multi-context system as an *information flow graph*. The information flow graph for example 4.3.1 is shown in figure 4.3.

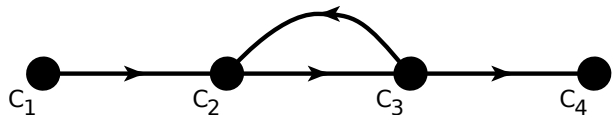

Figure 4.1: Information flow graph for the MCS in Example 4.3.1

A node (representing a context) is an *information provider* if it occurs as the first argument of the predicate *br* in any other context. A node is an *information receiver* if it contains any bridge rules. The information receivers and providers for each context are shown in the following table:

Table 4.1: Information receivers and providers for each context in Example 4.3.1

Context	Provider		Receiver
C1	null	null	{C2}
C2	C1	{b}	{C3}
	C3	{r}	
C3	C2	{q}	{C4}
C4	C3	{r} s	{}

For the flow of information to be automatically handled by the framework, each node in the system must be made aware of its information receivers so that changes in the contextual information can be selectively informed to latter. The `agent_initWFM` predicate, which takes the name of an agent, is provided for this purpose. Internally, this predicate scans the agent KB for br/2 terms, and sends a message to each provider agent, registering with it as a receiver. The terms a given agent is interested in is also bundled with the message. Below, we discuss the steps involved in computing the well-founded model.

To implement a multi-context system containing n contexts, the same number of agents must be created. Agents can be launched using `agent_create/4` predicate, providing `jcl.agent.ContextAgent.class` as the type argument. JCL relies on inter-agent message passing to set/update the list of information receivers and to automate the flow of information between agents. So `onMsgReceive(X)` predicate is triggered, only after ensuring that it is not for one of these purposes. Also, a *wfmManager* (instance of `jcl.context.WfmManager.class`) is attached to each instance of `jcl.agent.ContextAgent.class`. These are the only major differences between the `jal.agent.DefaultAgent.class` and `jcl.agent.ContextAgent.class`. The responsibilities of the *wfmManager* are discussed later.

In addition to the agents representing contexts, a special agent which communicates and coordinates the computation of a global well-founded model is required. This can be launched using the predicate `agent_create/4`, providing `jcl.agent.CoordinatorAgent.class` for the type argument. Unlike the agents we have discussed so far, this agent does not possess a KB or an inference engine, and is not a part of a MCS.

Before computing the global-well founded model, the coordinator agent must be aware of all the other agents in the system. So, it must be initialized using the predicate `agent_initWFM/1` (name of coordinator agent as the argument). Internally, this queries the JADE AMS agent for a list of other agents in the system.

After the initialization step, the predicate `computeGWFM/1`, which takes the name of the coordinating agent as the argument, can be used to start the process of computing the global model. The algorithm for computing the global well-founded model is shown in algorithm 1. The process of computing the reduct and closure is described later.

Algorithm 1: Computing the global well-founded model of a MCS

1: *set interpretation* $I = \{\}$
2: **while** !global fixpoint **do**
3: *compute the reduct of each context with respect to I*
4: **while** !global-closure **do**
5: *compute the closure of the reduct*

6: *send relevant parts of closure to information receivers*
7: *set I at each context to the most recent local closure*
8: **return** I

Starting with an empty interpretation, compute the reduct and closure at each node until an alternating fix point is reached.

In the first global iteration, the coordinator agent sends a message to all the MCS agents in the system, requesting them to compute the reduct of the logic programs contained in their KBs, with respect to an empty interpretation (I = {}).

The algorithm for computing the reduct is shown in algorithm 2. The knowledge base of each agent is a list of rules (`jpr.kb.Rule` objects), represented by $List_{rules}$ in algorithm 2. The body of each rule is a list of Prolog terms and so is an interpretation. For computing the reduct, we obtain a copy of rules contained in the agent KB and iterate through the body of each rule. If the *body* of a rule is empty, its *head* is added to a list of facts ($List_{redFacts}$). Else, the body of the rule is scanned for negated terms (not(p))) such that p is a part of the interpretation. If no such terms are found in a rule, it is added to a list of remaining rules ($List_{redRules}$). Otherwise, the rule is discarded. The $List_{redFacts}$ and $List_{redRules}$ lists are maintained by the wfmManager. These lists are updated each time a new reduct is computed.

Algorithm 1: Computing the reduct of a program

Input: $List_{rules}$ :: *knowledgebase*, $List_{Iterms}$ :: *interpretation*

Output(reduct): $List_{redFacts}$ :: *facts*, $List_{redRules}$

1: **for** each rule r in $List_rules$ **do**
2: **if** body of r is empty **then**
3: *add it to the* $List_{redFacts}$
4: **else**
5: *flag* $\leftarrow$ *true*
6: **for** each term t in body of r **do**

```
7:          if predicate of term t is not then
8:              t ← argument of t
9:              if t ϵ List_Iterms then
10:                 flag = false
11:                 break
12:             else
13:                 replace t by true
14:         if flag is true then
15:             add r to List_redRules
```

The reduct of each context in example 4.3.1 relative to an empty interpretation is shown below:

$$C_1 = \{b.,\ a :- b.)\}$$

$$C_2 = \{p :- br(a1, b).,\ q.\}$$

$$C_3 = \{r.\}$$

$$C_4 = \{w :- br(a3, r).,\ x :- w.\}$$

After computing the reduct, each agent in the system sends a message (in reply to "compute-reduct" message) to the coordinator agent with content as "reduct-computed". When all the agents in the system have responded successfully, the coordinator agent sends the following "compute-closure" message to all the agents in the system. This corresponds to step-5 in algorithm 1.

The receiving agents, using a fix point procedure shown in algorithm 3, compute the *closure* of their reduct and send the relevant terms to their information receivers. Each agent stores the incoming closure terms in a list of Prolog terms ($List_{bridgeFacts}$) maintained by the wfmManager. For example, consider two agents A_1 and A_2 corresponding to contexts C_1 and C_2 of example 4.3.1,

1. Lets say A_1 computes its closure as ({a,b}), and sends b to A_2.

2. On receiving the message from A_1, A_2 creates a new prolog term `br(a1,b)` and adds it to the *bridge facts*.

The $List_{bridgeFacts}$ list is empty for the first iteration.

Algorithm 3: Computing the closure of a reduct

Input: $List_{redFacts}$, $List_{redRules}$, $List_{bridgeFacts}$
Output: $List_{closure}$

```
1: while the length of closure list increases do
2:     List_closure ← List_redFacts
3:     List_closure ← List_bridgeFacts
4:     for each rule r in List_redRules do
5:         flag = true
6:         for each term t in body of r do
7:             if t is not true and t ∉ List_closure then
8:                 flag = false
9:                 break
10:        if flag == true then
11:            add head of r to List_closure
12: return List_closure
```

Add all the facts and the head of each satisfied rule to the closure list

As a response to the "compute-closure" message sent by the coordinator agent, each agent sends a message with content as "closure-computed", and an extra boolean field indicating if the most recent closure is same as the previously computed closure (if any). If any of the agents in the system respond with a *false*, the coordinating agent requests all the agents in the system to recompute their closure (and send it to their respective information receivers). Since the reduct contained in each agent KB is a definite logic program, the closure can only increase in size with addition of new information (bridgefacts). But as mentioned in section 2.4.1, this non-decreasing sequence has a fix-point. The system reaches a global closure if all the agents have reached their respective (local)definite fix points.

After a global closure is reached, the coordinating agent sends a compute-reduct message (with no interpretation attached) to all the agents in the system. In the second and all the following iterations, the reduct is computed with respect to the most recent definite fix point and bridge facts (interpretation $\leftarrow List_{closure} \cup List_bridgeFacts$). After computing the

reduct the *List_bridgeFacts* is cleared (empty). The process of computing a global closure repeats.

Let $dfp_{x,k}$ represent the k^{th} definite-fix-point reached by agent x. Algorithm 1 terminates when $dfp_{x,k-1}$ is equivalent to $dfp_{x,k+1}$ for some k and every 1≤x≤ n, where n is total number of MCS agents. Intuitively, when the definite fix-points reached by each agent alternate. The coordinator agent checks for the terminating condition by sending a message to the individual agents in the system each time a global closure is reached (after k=2). The global well-founded model is the union of the least definite-fix-points reached by each agent in the system. The global well-founded model computation for example 4.3.1 is shown in table 4.3.

Table 4.2: Computing the global well-founded model of MCS M in Example 4.3.1

Global Iter.	Local Iter.	C1	C2	C3	C4	Comments
1	0	{}	{}	{}	{}	I = {}
	1	{a,b}	{q}	{r}	{}	CR, CC, SC
	2	{a,b}	{p,q}	{r}	{w,x}	CC, SC
	3	{a,b}	{p,q}	{r}	{w,x}	GC
2	0	{a,b}	{b,p,q}	{q,r}	{r,w,x}	NI
	1	{a,b}	{}	{}	{}	CR, CC, SC
	2	{a,b}	{p}	{}	{}	CC, SC
	3	{a,b}	{p}	{}	{}	GC
3	0	{a,b}	{b,p}	{}	{}	NI
	1	{a,b}	{q}	{r}	{}	CR, CC, SC
	2	{a,b}	{p,q}	{r}	{w,x}	CC, SC
	3	{a,b}	{p,q}	{r}	{w,x}	GC

CR - Compute reduct with respect to most recent interpretation
CC - Compute closure of the most recent reduct
SC - Send the closure to information receivers
GC - Global closure
NI - New Interpretation

Summarizing, the whole process from a coordinator agent perspective is shown below:

```
1  Get a list of all the agents in the system and delete ams, df, and
   itself from the list.
```

```
2  Send a message to all the MCS agents, with "compute-reduct"
   as the content and interpretation I. I equals {} for the first
   global iteration.
3  Wait until all the MCS agents respond with "reduct-computed"
4  Send a message to all the MCS agents, with "compute-closure" as
   the content.
5  All the MCS agents respond with "closure-computed", and a boolean
   flag if the most recent closure has repeated.
6  If a global closure is reached, proceed else go to step-4.
7  Send a message to all the agents inquiring if a fix-point has been
   reached
7  If all the agents respond with a true, proceed, else go to step-2
8  End
```

4.4 A Loosely-Coupled System

The procedure outlined in the previous section computes the global well-founded model. But there are some limitations to this approach. In highly dynamic systems, the amount of time taken to compute the global well-founded model (reach an alternating fix point) could be (much) greater than the amount of time that could practically be allowed for reasoning. The primary issue is that non-monotonic reasoning works best for static systems or systems in which the change of knowledge over time is minimal. Moreover, the computation of the global model assumes that the agents are constantly connected to the coordinator agent. If any of the agents fail to respond/receive the *compute-reduct* or *compute-closure* messages, the computation is paused.

Therefore, we propose a loosely coupled system, where each agent (context) independently and locally computes a well-founded model and passes it to its information receivers. The timing of these events is entirely left to the application developer. Even if the local well-founded model computations are synchronized, the consequences are not guaranteed to match those of the equilibrium semantics (global well-founded model). This is evident from table 4.4, which shows the consequences of a synchronized loosely-coupled system.

However, though not guaranteed to, there could be a considerable amount of overlap between the consequences computed through this approach and the global well-founded

Table 4.3: The alternate method of computing semantics for MCS M in Example 4.3.1

Global Iteration	Local Iteration	C1	C2	C3	C4
1	0	{}	{}	{}	{}
	1	{a,b}	{q}	{r}	{}
	2	{a,b}	{q}	{r}	{}
2	0	{}	{}	{}	{}
	1	{a,b}	{p}	{}	{w,x}
	2	{a,b}	{p}	{}	{w,x}
3	0	{}	{}	{}	{}
	1	{a,b}	{p,q}	{r}	{}
	2	{a,b}	{p,q}	{r}	{}
4	0	{}	{}	{}	{}
	1	{a,b}	{p}	{}	{w,x}
	2	{a,b}	{p}	{}	{w,x}

model. This could be of interest to some applications. So the steps involved in this approach are outlined below. Clearly, there is no need for a coordinating agent.

1. All the agents in the system, except the coordinating agent, are created and launched.

2. The predicate `agent_intiWFM/1`, which takes the name of an agent as the argument, is used to update the list of receivers and senders.

3. The `computeWFM/2` predicate, which takes the name of an agent as the first argument, can be used to compute the local well-founded model, taking bridge facts into consideration. The result (list of Prolog terms) is bound to the second argument.

4. The `computeNsendWFM/1` predicate can be used to compute the model and send a copy of it to the information receivers.

4.5 Conclusion

In this chapter, we dealt with the task of ascribing semantics to the multi-agent reasoning system presented in chapter 3. Leveraging the inter-agent passing scheme, we computed the global well-founded model of a multi-context system. To our knowledge, there are no other existing implementations which serve the same purpose. We implemented a basic algorithm to compute the well-founded consequences. More optimized versions based on hypergraphs and tabling techniques are available [Berman et al., 1995; Zukowski et al., 1997; Niemelä and Simons, 1997; Rao et al., 1997] for computing the same, locally. The applicability of these techniques to a contextualized system should be investigated. The further research in this direction can test the viability of other logics.

Chapter 5

Conclusion

5.1 Contributions and Significance

In this chapter we reiterate the key contributions, significance and limitations of this research. This thesis presents the design and implementation of a reasoning framework for distributed knowledge management systems (DKMSs) on Android devices. We take an agent-based approach to model the constituent elements of a DKMS. The JPR Agent Library facilitates the creation and maintenance of reasoning agents directly from Java Prolog Reasoner (JPR). JPR is an implementation of the Prolog programming language designed to make Prolog usable on Android devices. The agent library provides predicates for the following purposes.

- To create and terminate agents directly from JPR. Each agent with its own knowledge base (KB) and inference engine acts as an autonomous reasoning entity.
- To load Prolog files to an agent KB
- To query an agent
- To trigger the behaviors(actions) of an agent from Prolog
- To send or receive messages from other agents (in the distributed system)

An Android application demonstrating the *ease-of-use* of this library was also presented. A list of predicates defined in the library is provided in appendix-A.

The latter half of this thesis deals with the problem of assigning semantics to this distributed system. The multi-context systems framework [Giunchiglia and Serafini, 1994;

Brewka et al., 2007, 2011; Brewka, 2013; Serafini and Homola, 2012] is a prominent theoretical framework in AI for contextual reasoning. To adopt this framework, each agent KB, containing a normal logic program, is treated as a *context*. The JPR context library provides predicates using which

- the interrelations between various agents KBs(bridge rules) can be declaratively specified, and
- a global well-founded model of the system can be computed.

. The global well-founded model corresponds to the equilibrium semantics of the non-monotonic multi-context systems framework proposed by Brewka et Al. [2011]. Additionally, a loosely-coupled mode of execution is proposed for highly dynamic systems.

So far, the context-related research in AI has been largely theoretical with only a few implementations. Even in the few implementations that do exist, there is hardly any discussion on applicability for mobile devices. Furthermore, the research in context-aware systems is gaining momentum due to the advances in sensing capabilities of mobile devices. But again, to our knowledge, there is hardly any framework for developing rule-based applications on mobile devices. This research helps fill these gaps by providing a software framework for contextual reasoning on Android devices. This framework, in conjunction with the JPR Sensor Library, can be used to develop rule-based context-aware applications on Android devices.

5.2 Future Research

Since this effort is only a first step towards mobile-based contextual reasoning, there is plenty of scope for further research. Below, we provide some pointers.

- Optimization: In the past, some optimization schemes for local well-founded model computations were proposed [Berman et al., 1995; Brass et al., 2001]. Also, topology-based optimization techniques for computing the equilibria of a multi-context system

were proposed. Since the target devices are resource-constrained, the applicability of these optimization schemes must be investigated.

- Grounding and forward chaining: Forward chaining rule-based systems are highly suitable for event handling in context-aware systems. Also, the well-founded model computation requires ground logic programs. Since programs with function symbols cannot be ground, alternate techniques are required. We intend to provide a reasonable way of handling these issues in the next version of this framework.

- Applicability of other logics for contexts: The applicability of logics other than well-founded semantics for logic programs has to be tested. To avail this, we have designed this framework in a modular fashion.

Appendix A

Disaster Response Application

Here we provide a detailed description of the Disaster Management Application developed as a part of this thesis. This is intended to serve as a demo application for the developers using the JPR Agent Library discussed in Chapter 2. Several ways of configuring a disaster response system were discussed in the same chapter. A summary of the configuration we end up pursuing is given below:

A disaster control center is a command and control facility responsible for planning and monitoring the response activities in a disaster-hit area. The aim of this application(s) is to provide tools for knowledge management in such situations. It is assumed that a disaster control center (DCC) host a PC/server and that all the personnel taking part in the response activities are carrying an Android device.

In the initial stages of the disaster response, responders and volunteers collect data from the affected area that depicts the severity of the situation such as number of victims affected, food and shelter required etc. This data must be passed to the disaster control center for planning, managing and distributing resources, etc. Moreover, it is important that the data collected is labeled by its source. The activity data (%split of the duration a volunteer/responder) was standing, sitting or walking would indicate the physical strain he/she is subjected to. The tasks assigned on the following day can be planned accordingly (avoiding fatigue). If the responder/volunteer needs to approach someone for help, he must know the location of the nearest warehouse and/or emergency responder. Below, we discuss the basic steps involved in designing an Android application that can fulfill the requirements discussed

so far. These steps can guide the development of similar applications. Prolog files specific to this application are provided at the end of this chapter[1].

To implement this, a main-container can be launched on the DCC-PC by following the steps outlined below [2]:

- On the JAVA side, create a Prolog console (jpr.PrologConsole.class) which has an associated inference engine.
- Load the agent library to the inference engine wrapped by the console.
- Create a main-container providing the IP address of the machine and a suitable port, using the `mainContainer_create/4` predicate
- Create an agent named 'dccAgent', using the `agent_create` predicate.
- Create a behavior file (.pl file) containing the behaviors of the dccAgent. In the simplest case, the dccAgent only needs to respond to incoming messages. For this, the developer only has to write the body of the rule '`onMsgReceive(X):-` '.
- Load this file to the dccAgent, using the `behaviorFile_load/2` predicate.
- Issue the query 'listing. ', to the agent to double-check if all the above steps were successful.
  `agent_query(dccAgent,'listing. ').`

Similarly for the volunteer side, an Android application is created with a blank activity called MainActivity.class, following the standard steps in the developer guide[3]. A snapshot of this activity is shown in figure A.1.

The `jade.android.MicroRuntimeService` is declared in the manifest file of the application. The following steps are performed in the startup of the disaster response application (main activity):

1. A form is displayed, requesting the volunteer to enter his name.

[1] @maier: to be added
[2] A list of predicates is provided in the Appendix-B for reference.
[3] https://developer.android.com/guide/index.html

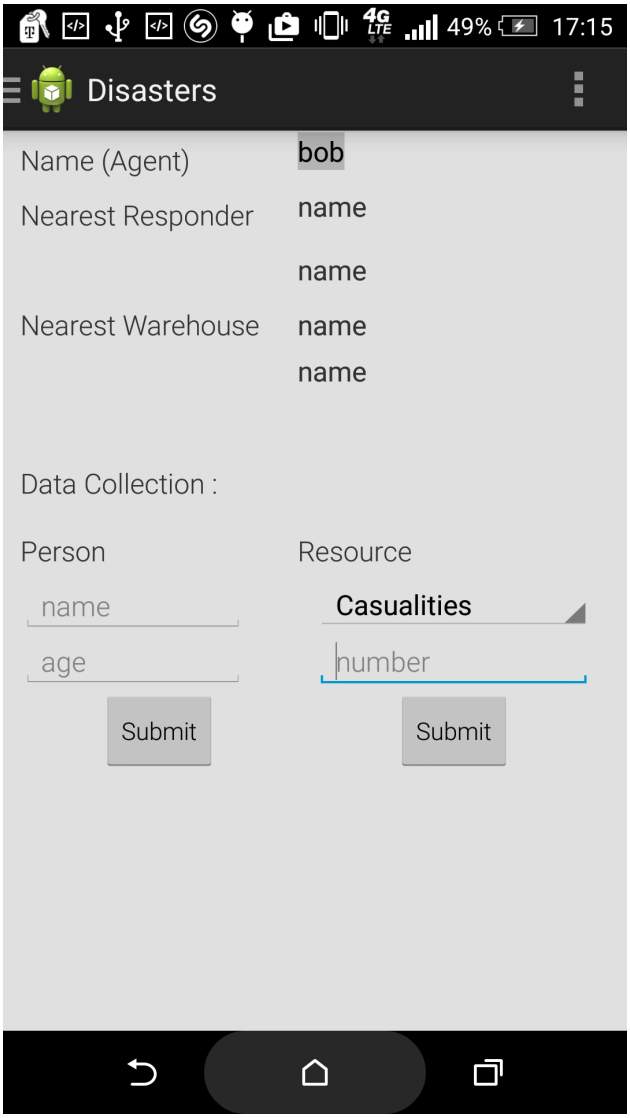

Figure A.1: The main activity with disaster related data

2. An instance of JPR is created. An instance of the agent library is created by passing the context of the application and *true*(for isSplit) as parameters. This is loaded to JPR using the `loadLibrary` method.

3. A split container is created and an agent is launched with the same name as the volunteer.

4. The behavior file of this agent is loaded.

The behavior file of the agent for this application must contain rules which can be triggered in response to button clicks. For example, if the volunteer fills a form, using the drop-down list,text fields etc. and clicks the *submit* button, this should fire a rule which in its body sends this data to the *dccAgent*. The agents can also send the GPS location when requested by remote agents. The GPS information would be highly useful to plan collaborative tasks.

Appendix B

List of Predicates

Predicate	Arguments	Function
mainContainer_create/4	Host,Port,Name of the container,Variable	To create a main container using the arguments provided
mainContainer_create/3	Host,Port, Variable	To create a main container with the default name main-container
isMain/1	Variable	To retrieve the main container (reference term)
container_create/4	Name of the container,Host,Port,Variable	To create an agent container, store object reference of its controller in a reference term
list_containers/1	Variable	Returns a list of active containers
mtp_add/2	Name of the container,IPAddress	Adds an MTP to a container
mtp_del/2	Name of the container,IPAddress	Removes an existing MTP
splitContainer_create/3	Name of the container,Address and Port of main-container	Creates a split-container on Android device
fullContainer_create/3	Name of the container,Address and Port of main-container	Creates a full container on Android device
fullMainContainer_create/3	Name of the container,Address and Port of main-container	Creates a main container on an Android device
container_kill/0		Terminates a running container on an Android device
agent_create/4	Name of the agent, (path to) Class of the agent,Name of the container &Variable	To create an agent on a PC

agent_create/2	Name of the agent, Class of the agent	To create an agent on an Android device
agent_kill/1	Name of the agent	To terminate an agent
agent_container/2	Name of the Agent, Name of the container	To store agent-container pairs
agent_query/3	Name of the agent, Query, Asynchronous or not	To query an agent on the device from the device IE
agent_message/3	Sender, Receiver, Message	To request the Sender to send the Message (String) to the Receiver
agent_changeStateTo/1	New state (activate/suspend)	To switch between activated and suspended
oneShotBehavior_add/2	Name of the agent,agent behavior to be loaded	To trigger a predicate defined in the agent behavior file as a one shot behavior
cyclicBehavior_add/2	Name of the agent,agent behavior to be loaded	To trigger a predicate defined in the agent behavior file as a cyclic behavior
wakerBehavior_add/3	Name of the agent,agent behavior to be loaded, time delay	To trigger a predicate defined in the agent behavior file as a waker behavior.
tickerBehavior_add/3	Name of the agent,agent behavior to be loaded, time delay	To trigger a predicate defined in the agent behavior file as a ticker behavior.
agent_getCurrBehav/1	Name of the agent	Returns a prolog list of behaviors in the agent queue
agent_removeBehav/2	Name of the agent, Name of the behavior	To remove a behavior from the agent queue
assert_br/1	Rule	To assert a bridge rule to an agent KB
retract_br/1	Rule	To retract a bridge rule from an agent KB
init_wfm/0		To initialize the information receivers and providers
info_receivers/1	Variable	Returns a list of information receivers
info_providers/1	Variable	Returns a list of information providers

lwfm/1	Variable	Returns the local well-founded model of an agent KB

Bibliography

Gregory D. Abowd. Classroom 2000: An experiment with the instrumentation of a living educational environment. *IBM systems journal*, 38(4):508–530, 1999.

Gregory D Abowd, Christopher G Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless networks*, 3(5):421–433, 1997.

Gediminas Adomavicius and Alexander Tuzhilin. Context-aware recommender systems. In *Recommender systems handbook*, pages 217–253. Springer, 2011.

Robert M Akscyn, Donald L McCracken, and Elise A Yoder. Kms: a distributed hypermedia system for managing knowledge in organizations. *Communications of the ACM*, 31(7):820–835, 1988.

Grigoris Antoniou, Constantinos Papatheodorou, and Antonis Bikakis. Reasoning about context in ambient intelligence environments: A report from the field. In *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, 2010.

Krzysztof R Apt and Roland N Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19:9–71, 1994.

Costin Bădică, Adriana Bădită, and Maria Ganzha. Implementing rule-based mechanisms for agent-based price negotiations. In *Proceedings of the 2006 ACM symposium on applied computing*, pages 96–100. ACM, 2006.

Costin Badica, Lars Braubach, and Adrian Paschke. Rule-based distributed and agent systems. In *Rule-Based Reasoning, Programming, and Applications*, pages 3–28. Springer, 2011.

Matteo Baldoni, Cristina Baroglio, Viviana Mascardi, Andrea Omicini, and Paolo Torroni. Agents, multi-agent systems and declarative programming: what, when, where, why, who, how? In *A 25-year perspective on logic programming*, pages 204–230. Springer-Verlag, 2010.

Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009.

Aaron Beach, Mike Gartrell, Xinyu Xing, Richard Han, Qin Lv, Shivakant Mishra, and Karim Seada. Fusing mobile, sensor, and social data to fully enable context-aware computing. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, pages 60–65. ACM, 2010.

Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with jade. In *Intelligent Agents VII Agent Theories Architectures and Languages*, pages 89–103. Springer, 2001.

Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

Kenneth A Berman, John S Schlipf, and John V Franco. *Computing the well-founded semantics faster*. Springer, 1995.

Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, 2010.

Antonis Bikakis and Grigoris Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *Knowledge and Data Engineering, IEEE Transactions on*, 22(11):1492–1506, 2010a.

Antonis Bikakis and Grigoris Antoniou. Rule-based contextual reasoning in ambient intelligence. In *Semantic Web Rules*, pages 74–88. Springer, 2010b.

VG Bogdanova, IV Bychkov, AS Korsukov, GA Oparin, and AG Feoktistov. Multiagent approach to controlling distributed computing in a cluster grid system. *Journal of Computer and Systems Sciences International*, 53(5):713–722, 2014.

Matteo Bonifacio, Paolo Bouquet, Gianluca Mameli, and Michele Nori. Kex: a peer-to-peer solution for distributed knowledge management. In *Practical Aspects of Knowledge Management*, pages 490–500. Springer, 2002a.

Matteo Bonifacio, Paolo Bouquet, and Paolo Traverso. Enabling distributed knowledge management: Managerial and technological implications. 2002b.

Rafael H Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J Gomez-Sanz, Joao Leite, Gregory MP O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

Paolo Bouquet, Fausto Giunchiglia, Frank Van Harmelen, Luciano Serafini, and Heiner Stuckenschmidt. C-owl: Contextualizing ontologies. In *The Semantic Web-ISWC 2003*, pages 164–179. Springer, 2003.

Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 1(05):497–538, 2001.

Gerhard Brewka. Towards reactive multi-context systems. In *Logic Programming and Nonmonotonic Reasoning*, pages 1–10. Springer, 2013.

Gerhard Brewka and Thomas Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI*, volume 7, pages 385–390, 2007.

Gerhard Brewka, Floris Roelofsen, and Luciano Serafini. Contextual default reasoning. In *IJCAI*, pages 268–273, 2007.

Gerhard Brewka, Thomas Eiter, and Michael Fink. Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In *Logic programming, knowledge representation, and nonmonotonic reasoning*, pages 233–258. Springer, 2011.

Rodney A Brooks. How to build complete creatures rather than isolated cognitive simulators. *Architectures for intelligence*, pages 225–239, 1991.

Bruce G Buchanan and Edward A Feigenbaum. Dendral and meta-dendral: Their applications dimension. *Artificial intelligence*, 11(1):5–24, 1978.

Hung H Bui, Svetha Venkatesh, and Geoff West. Tracking and surveillance in wide-area spatial environments using the abstract hidden markov model. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(01):177–196, 2001.

Saga BuvaE and Ian A Mason. Propositional logic of context. In *Proceedings of the eleventh national conference on artificial intelligence*, 1993.

Bo Chen, Harry H Cheng, and Joe Palen. Mobile-c: a mobile agent platform for mobile c/c++ agents. *Software: Practice and Experience*, 36(15):1711–1733, 2006.

Harry Chen, Tim Finin, and Amupam Joshi. Semantic web in the context broker architecture. Technical report, DTIC Document, 2005.

Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a context-aware tourist guide: the guide project. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 20–31. ACM, 2000.

Krzysztof Chmiel, Maciej Gawinecki, Pawel Kaczmarek, Michal Szymczak, and Marcin Paprzycki. Efficiency of jade agent platform. *Scientific Programming*, 13(2):159–172, 2005.

Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1978.

Dustin Cline. Integrating logic programming with description logic reasoning and sensor observation management for mobile devices. Master's thesis, University of Georgia, 2015.

A Colmeraner, Henri Kanoui, Robert Pasero, and Philippe Roussel. Un systeme de communication homme-machine en francais. Luminy, 1973.

Laura Daniele, Patrícia Dockhorn Costa, and Luís Ferreira Pires. Towards a rule-based approach for context-aware applications. In *Dependable and Adaptable Networks and Services*, pages 33–43. Springer, 2007.

Minh Dao-Tran. *Multi-Context Systems: Algorithms and Efficient Evaluation*. PhD thesis, Vienna University of Technology, March 2014.

Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. The semantic web: The roles of xml and rdf. *Internet Computing, IEEE*, 4(5):63–73, 2000.

Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

Anind K Dey. Modeling and intelligibility in ambient environments. *Journal of Ambient Intelligence and smart environments*, 1(1):57–62, 2009.

Charalampos Doukas, Ilias Maglogiannis, Philippos Tragas, Dimitris Liapis, and Gregory Yovanof. Patient fall detection using support vector machines. In *Artificial Intelligence and Innovations 2007: from Theory to Applications*, pages 147–156. Springer, 2007.

Weichang Du and Lei Wang. Context-aware application programming for mobile devices. In *Proceedings of the 2008 C 3 S 2 E conference*, pages 215–227. ACM, 2008.

Amal El Fallah-Seghrouchni and Alexandru Suna. Claim: A computational language for autonomous, intelligent and mobile agents. In *Programming Multi-Agent Systems*, pages 90–110. Springer, 2004.

Richard Etter, Patricia Dockhorn Costa, and Tom Broens. A rule-based approach towards context-aware user notification services. In *Pervasive Services, 2006 ACS/IEEE International Conference on*, pages 281–284. IEEE, 2006.

FIPA. Fipa, abstract architecture specification. *Foundation for Intelligent Physical Agents,http://www.fipa.org/specs/fipa00001/SC00001L.html (5.10.2015)*, 2002.

ACL Fipa. Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00061/SC00061G.html (5.10.2015)*, 2002a.

ACL Fipa. Fipa agent management specification. *Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00023/SC00023K.html (5.10.2015)*, 2002b.

Ernest Friedman-Hill. *JESS in Action*. Manning Greenwich, CT, 2003.

Les Gasser. Mas infrastructure definitions, needs, and prospects. In *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, pages 1–11. Springer, 2001.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

Michael P Georgeff and Amy L Lansky. Reactive reasoning and planning. In *AAAI*, volume 87, pages 677–682, 1987.

Chiara Ghidini and Fausto Giunchiglia. Local models semantics, or contextual reasoning= locality+ compatibility. *Artificial intelligence*, 127(2):221–259, 2001.

Joseph C Giarratano et al. Clips user's guide. *NASA Technical Report, Lyndon B Johnson Center*, 1993.

Fausto Giunchiglia and Luciano Serafini. Multilanguage hierarchical logics, or: how we can do without modal logics. *Artificial intelligence*, 65(1):29–70, 1994.

Guido Governatori, Michael J Maher, Grigoris Antoniou, and David Billington. Argumentation semantics for defeasible logic. *Journal of Logic and Computation*, 14(5): 675–702, 2004.

Ramanathan V Guha. *Contexts: a formalization and some applications*, volume 101. Stanford University Stanford, CA, 1991.

Karen Henricksen and Jadwiga Indulska. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and mobile computing*, 2(1): 37–64, 2006.

Koen V Hindriks, Frank S De Boer, Wiebe Van der Hoek, and John-Jules Ch Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

Jongyi Hong, Eui-Ho Suh, Junyoung Kim, and SuYeon Kim. Context-aware system for proactive personalized service based on context history. *Expert Systems with Applications*, 36(4):7448–7457, 2009.

Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*, 2001.

Zakwan Jaroucheh, Xiaodong Liu, and Sally Smith. Recognize contextual situation in pervasive environments using process mining techniques. *Journal of Ambient Intelligence and Humanized Computing*, 2(1):53–69, 2011.

Nicholas R Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

Maarit Kangas, Antti Konttila, Ilkka Winblad, and Timo Jamsa. Determination of simple thresholds for accelerometry-based parameters for fall detection. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*, pages 1367–1370. IEEE, 2007.

Michael Knappmeyer, Nigel Baker, Saad Liaquat, and Ralf Tönjes. A context provisioning framework to support pervasive and ubiquitous applications. In *Smart Sensing and Context*, pages 93–106. Springer, 2009.

Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974.

Kenneth Kunen. Negation in logic programming. *The Journal of Logic Programming*, 4(4):289–308, 1987.

Xining Li. Imago: A prolog-based system for intelligent mobile agents. In *Mobile Agents for Telecommunication Applications*, pages 21–30. Springer, 2001.

Seng W Loke. Logic programming for context-aware pervasive computing: Language support, characterizing situations, and integration with the web. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 44–50. IEEE Computer Society, 2004.

Faisal Luqman and Martin Griss. Overseer: a mobile context-aware collaboration and task management system for disaster response. In *Creating Connecting and Collaborating through Computing (C5), 2010 Eighth International Conference on*, pages 76–82. IEEE, 2010.

Michael J Maher. A model-theoretic semantics for defeasible logic. *arXiv preprint cs/0207086*, 2002.

Frederick Maier. A study of defeasible logics. 2007.

Frederick Maier and Donald Nute. Well-founded semantics for defeasible logic. *Synthese*, 176(2):243–274, 2010.

John McCarthy. *LISP 1.5 programmer's manual*. MIT press, 1965.

John McCarthy. Circumscriptiona form of non-monotonic reasoning. *Artificial intelligence*, 13(1):27–39, 1980.

John McCarthy. Generality in artificial intelligence. *Communications of the ACM*, 30 (12):1030–1035, 1987.

John McCarthy. Artificial intelligence, logic and formalizing common sense. In *Philosophical Logic and Artificial Intelligence*, pages 161–190. Springer, 1989.

John McCarthy. Notes on formalizing context. 1993.

Jack Minker. An overview of nonmonotonic reasoning and logic programming. *The Journal of Logic Programming*, 17(2):95–126, 1993.

Marvin Minsky. A framework for representing knowledge. 1974.

Assaad Moawad, Antonis Bikakis, Patrice Caire, Grégory Nain, and Yves Le Traon. A rule-based contextual reasoning platform for ambient intelligence environments. In *Theory, Practice, and Applications of Rules on the Web*, pages 158–172. Springer, 2013.

David Morley and Karen Myers. The spark agent framework. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 714–721. IEEE Computer Society, 2004.

Grzegorz J Nalepa and Szymon Bobek. Rule-based solution for context-aware reasoning on mobile devices. *Computer Science and Information Systems*, 11(1):171–193, 2014.

Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

Ilkka Niemelä and Patrik Simons. Smodelsan implementation of the stable model and well-founded semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer, 1997.

Donald Nute. Defeasible logic. *Handbook of logic in artificial intelligence and logic programming*, 3:353–395, 1994.

MN Nyan, Francis EH Tay, and E Murugasu. A wearable system for pre-impact fall detection. *Journal of biomechanics*, 41(16):3475–3481, 2008.

Stefan Poslad. Specifying protocols for multi-agent systems interaction. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 2(4):15, 2007.

Teodor C Przymusinski. Non-monotonic formalisms and logic programming. In *ICLP*, pages 655–674. Citeseer, 1989.

Prasad Rao, Konstantinos Sagonas, Terrance Swift, David S Warren, and Juliana Freire. Xsb: A system for efficiently computing well-founded semantics. In *Logic Programming And Nonmonotonic Reasoning*, pages 430–440. Springer, 1997.

Raymond Reiter. *On closed world data bases*. Springer, 1978.

Raymond Reiter. A logic for default reasoning. *Artificial intelligence*, 13(1):81–132, 1980.

Daniele Riboni and Claudio Bettini. Cosar: hybrid reasoning for context-aware activity recognition. *Personal and Ubiquitous Computing*, 15(3):271–289, 2011.

Daniel Salber, Anind K Dey, and Gregory D Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 434–441. ACM, 1999.

M Mohsin Saleemi, Natalia Díaz Rodríguez, Johan Lilius, and Iván Porres. A framework for context-aware applications for smart spaces. In *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 14–25. Springer, 2011.

Andrea Santi, Marco Guidi, and Alessandro Ricci. Jaca-android: An agent-based platform for building smart mobile applications. In *Languages, Methodologies, and Development Tools for Multi-Agent Systems*, pages 95–114. Springer, 2011.

Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90. IEEE, 1994.

John R Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

Luciano Serafini and Martin Homola. Contextualized knowledge repositories for the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 12:64–87, 2012.

Herbert Alexander Simon. *The sciences of the artificial*, volume 136. MIT press, 1969.

Thomas Strang and Claudia Linnhoff-Popien. A context modeling survey. In *Workshop Proceedings*, 2004.

Arnon Sturm and Onn Shehory. The evolution of mas tools. In *Agent-Oriented Software Engineering*, pages 275–288. Springer, 2014.

Chuan-Jun Su and Chia-Ying Wu. Jade implemented mobile multi-agent based, distributed information platform for pervasive health care monitoring. *Applied Soft Computing*, 11(1):315–325, 2011.

Michael Thielscher. Flux: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.

Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

Allen Van Gelder, Kenneth A Ross, and John S Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM (JACM)*, 38(3):619–649, 1991.

Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems (TOIS)*, 10(1):91–102, 1992.

Michael Winikoff. Jack intelligent agents: An industrial strength platform. In *Multi-Agent Programming*, pages 175–193. Springer, 2005.

Michael Wooldridge and Nicholas R Jennings. Agent theories, architectures, and languages: a survey. In *Intelligent agents*, pages 1–39. Springer, 1995.

Zhiming Zhao, Adam Belloum, Cees De Laat, Pieter Adriaans, and Bob Hertzberger. Using jade agent framework to prototype an e-science workflow bus. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 655–660. IEEE, 2007.

Ulrich Zukowski, Stefan Brass, and Burkhard Freitag. Improving the alternating fixpoint: The transformation approach. In *Logic Programming and Nonmonotonic Reasoning*, pages 40–59. Springer, 1997.