Distributed Abductive Reasoning System
and
Abduction In the Small

Author:
Jiefei Ma

Supervisors:
Dr. Krysia Broda, Dr. Alessandra Russo

June 22, 2007
http://www.doc.ic.ac.uk/~jm103/project
Abstract

Abductive reasoning (abduction) is a powerful reasoning method that can be used in many applications such as fault diagnosis, medical diagnosis and automated planning. Recently, it also has attracted many interests from the world of ubiquitous computing – Tiny devices are everywhere. How can we have small devices with limited computational power and space to perform the abductive reasoning tasks? Also, can we have a collection of devices to perform abductive reasoning collaboratively?

The traditional proof procedures of abductive reasoning are computationally hungry. In order to have small devices to do abduction, we need to have an efficient enough implementation of the proof procedures to run on them. This may involves various optimisations. To have a collection of devices to collaborate in reasoning tasks is an even harder problem – How can we make sure the reasoning result is consistent among the devices?

This report first describes how to implement the Kakas-Mancorella proof procedure for abductive reasoning efficiently and to run it on a tiny Gumstix computer. The report then describes the development of a multi-agent system DARES (Distributed Abductive REasoning System). DARES allows a group of agents to collaborate in abductive reasoning tasks – it allows the agents to seek help form others during the reasoning process and guarantees the results obtained are globally consistent among the agents. DARES also provides an efficient mechanism for the agents to exchange tasks and results among themselves. The report finally describes how to deploy DARES to the Gumstix computers.
Acknowledgements

First of all, I would like to thank my supervisors, Dr. Krysia Broda and Dr. Alessandra Russo, for their enormous support and encouragement during my project. They have always been excited about my work and have given me lots of valuable inspirations.

Secondly, I would like to thank Dr. Emil Lupu and Professor Morris Sloman for the ideas about the applications for the project and providing me the Gumstix computers for the project.

I would like to my personal tutor, Professor Susan Eisenbach, for introducing me to my supervisors.

I also would like to thank the the following people:

- Professor Murray Shanahan for discussion and ideas about abductive planning.
- Professor Keith Clark for the discussion and ideas about multi-agent systems.
- Peter Robinson, the author of QuProlog, for supporting my evaluation of Pedro and QuProlog.
- Peter Collingbourne for giving me help during the initial porting of YAP Prolog.

Finally, I would like to thank my parents who have sponsored me to study in the UK for the past 6 years, and thank my girl friend Sylvaine for looking after me while I was too busy with my project.
# Contents

1. **Introduction**  
   1.1 Objectives .................................................. 3  
   1.2 Contributions .............................................. 3  

2. **Background**  
   2.1 Logic Programming ........................................... 5  
      2.1.1 Syntax and Informal Semantics of Prolog .......... 5  
      2.1.2 Query Evaluation in Prolog ......................... 7  
      2.1.3 Prolog Compilers and the Warren Abstract Machine 10  
   2.2 Partial Evaluation in Logic Programs ..................... 15  
      2.2.1 Partial Evaluation .................................. 15  
      2.2.2 A Meta-Interpreter of Prolog ...................... 17  
      2.2.3 A Partial Evaluated Meta-Interpreter .............. 18  
   2.3 Abductive Reasoning ....................................... 20  
      2.3.1 Logical Reasoning ................................... 20  
      2.3.2 Formal Definition .................................. 20  
      2.3.3 Proof Procedure ................................... 21  
      2.3.4 Applications ....................................... 23  
   2.4 Event Calculus and Abductive Planning ..................... 24  
      2.4.1 Introduction to Event Calculus .................... 24  
      2.4.2 Planning with Abductive Reasoning ................. 25  
   2.5 Agent Communications in Multi-Agent Systems .......... 25  
      2.5.1 KOMI ............................................... 26  
      2.5.2 Mediation Services with KOMI ..................... 27  
   2.6 Abductive Logic Agent System – ALIAS .................... 28  
      2.6.1 The Architecture ................................... 29  
      2.6.2 The LAILA Language ................................ 29  
      2.6.3 Proposed Applications ............................... 31  
   2.7 Target Device – Gumstix ................................... 31  
      2.7.1 Specification ....................................... 32  
      2.7.2 Resources ........................................ 33  
   2.8 QuProlog .................................................... 33  
      2.8.1 Multi-threads Support ................................ 34  
      2.8.2 Inter-agent Communications Model (ICM) .......... 34  
      2.8.3 Pedro ............................................. 35
## Contents

### 3 Investigation of Abductive Reasoning

3.1 Abductive Meta-Interpreter ........................................ 36
   3.1.1 Re-engineering the Kakas’ ALP Meta-Interpreter ............ 36
   3.1.2 Benchmarking .................................................. 39
   3.1.3 Running On a Gumstix ......................................... 40

3.2 Optimisation by Partial Evaluation ............................... 40
   3.2.1 Observations of the Abductive Meta-Interpreter ............ 40
   3.2.2 Compiling Away the Meta-Interpreter ....................... 41
   3.2.3 Benchmarking .................................................. 44
   3.2.4 Conclusion .................................................... 45

### 4 Distributed Reasoning System ................................. 46

4.1 Overview .................................................................. 46
   4.1.1 Motivation ......................................................... 46
   4.1.2 Limitations of ALIAS ........................................... 47
   4.1.3 Basic Assumptions .............................................. 47
   4.1.4 Desired Properties ............................................. 48

4.2 System Version 1 .................................................... 49
   4.2.1 Architecture Overview ......................................... 49
   4.2.2 Inter-agent Communication .................................... 51
   4.2.3 The Central Directory Server ................................. 52
   4.2.4 Reasoning Agents ............................................... 55

4.3 System Version 2 .................................................... 61
   4.3.1 Architecture Overview ......................................... 61
   4.3.2 Inter-agent Communication .................................... 61
   4.3.3 Reasoning Agents ............................................... 63

4.4 Comparison of Two Versions ....................................... 64
   4.4.1 Conclusion ....................................................... 65

4.5 Example of Multi-Agent Reasoning ............................... 65
   4.5.1 A Distributed Deductive Meta-Interpreter ................. 65
   4.5.2 The Sample Database – Family Tree ......................... 68

### 5 Distributed Abductive Meta-Interpreters ....................... 69

5.1 General Purpose Distributed Abductive Meta-Interpreter .... 69
   5.1.1 Scenario ......................................................... 69
   5.1.2 Proof Procedure ................................................ 70
   5.1.3 Correctness of the Procedure ................................ 74
   5.1.4 Optimisation .................................................... 75

5.2 Distributed Abductive Event Calculus Planner ................. 76
   5.2.1 Scenario ......................................................... 76
   5.2.2 Planner as Meta-Interpreter ................................ 79
   5.2.3 Optimisations .................................................. 83

### 6 Porting to the Gumstix .......................................... 84

6.1 Cross Compiling the Prolog Engines ............................ 84
   6.1.1 Selecting the Suitable Prolog Engines ...................... 85
   6.1.2 The Cross-Compilation Process .............................. 86
   6.1.3 Network Setup .................................................. 88

6.2 Testing and Benchmarking ........................................ 89
CONTENTS

7 Conclusion 91
  7.1 Summary ................................................................. 91
  7.2 Further Work ............................................................ 92
    7.2.1 Optimisations and Extensions of DARES ......................... 92
    7.2.2 Identify Potential Applications ................................. 93
  7.3 Final Thoughts ........................................................... 93

A Sample Code 96
  A.1 Kakas ALP Meta-Interpreter ........................................ 96
  A.2 The Distributed Family Tree Database ............................. 100
Chapter 1

Introduction

Abductive Reasoning (Abduction) is a method of reasoning and has been studied for a while. It has a number of proposed applications such as fault diagnosis, medical diagnosis and automated planning.

Though the applications have been investigated and some prototypes have been created, the use of abduction is still at a theoretical or experimental stage. The first difficulty of applying abduction is how to represent the knowledge in a declarative means and the second difficulty is how to reasoning from the knowledge efficiently. There has been a number of abductive proof procedures proposed over the years and implemented in a logic programming language (i.e. Prolog). Some implementations of the proof procedures (e.g. Kakas-Mancarella procedure[17]) are tested on modern standard PC and have given promising results.

However, are they really practical enough for real world pervasive computing applications? For example, we have a few Gumstix computers, each of which is as small as a chewing gum and has a 400MHz processor and only 64MB memory. Can we have an efficient enough implementation of any abductive proof procedure to turn these small yet functional devices into intelligent agents that can perform abductive reasoning?

Another interesting problem that attracts more and more attentions is how to have a set of intelligent agents to collaborate on an abductive reasoning task? Consider the following scenario (three fellows are having a conversation):

A : How about having steamed rice tonight?
C : Nice! But do you know how to make streamed rice?
A : Yes, it is very easy, we just need a rice cooker and the rice. But where do we get the cooker from?
B : How about we buy it from Argos?
A : OK. But What about the rice?
C : I know Tesco sells rice. So we can buy the rice there.
A : Sounds good! But.. We don’t have enough money to buy both the cooker and the rice. What else can we do?
B : Hmm, I may be able to borrow the rice cooker from one of my friends.
A : And the rice?
C: Buy it from Tesco!
A: Great! That’s the plan then.

In the above example (Figure 1.1), each of the fellows has his/her life experiences and they collaborated together to find a plan for making steamed rice. At first, they came up with plan “buy cooker and buy rice”, but the plan was later rejected one of them. In the second attempt, they found another plan “borrow cooker and buy rice”. This time the plan was agreed by all of fellows so they decided to take it.

If we have a group of loosely coupled agents, each of which has its own knowledge including the integrity constraints, how can we coordinate them to work collaboratively on a reasoning task? One may suggest that we can put together what all the agents know about, and then perform a monolithic abductive reasoning from it. However, this is not always feasible. For example, an agent may not be able to tell you everything it knows when it is asked, or simply the “merged” knowledge base is too big for anyone to “remember” it (e.g. lack of memory space) or to reason from it. Also, what happens when the group of agents changes?

When merging the knowledge is not feasible, the agents may still be able to collaborate on an abductive reasoning task. For example, given a goal, an agent can try to perform reasoning on its own (i.e. using its own knowledge). If it is stuck on resolving a sub-goal, it may ask another agent for help. When the result for the sub-goal, if any, is returned, the agent can use it and continue the reasoning process.

However, there are three issues need to be addressed:

- How does an agent decide when and where to ask for help?
- When a result is returned from a set of agents, how can we make sure the result is
indeed agreed by all the agents in the set (i.e. it does not violate any of the agents’ integrity constraints)?

- Again, how practical is it?

If the above problems can be resolved and such a distributed abductive reasoning framework/system is developed, it will benefit the world of pervasive computing. For example, a collection of body sensors may be able to reasoning about the monitored data collaboratively and hence decide a suitable action (e.g. raising an alarm).

The topic of this project is firstly to investigate how to implement the Kakas-Mancarella procedure efficiently for running on the Gumstix computers, and the possibility of optimising it by partial evaluation. Then, to resolve the issues raised for distributed abductive reasoning, design and implement a multi-agent system for a set of abductive agents to perform reasoning consistently. The final but ambitious goal is to deploy a collection of Gumstix computers into set of intelligent abductive reasoning agents by running the developed system on it.

1.1 Objectives

The contributions of this project include:

**Abduction In the Small** which is to run abduction on the Gumstix computers. This involves the following tasks:

1. To develop an efficient implementation of the Kakas-Mancarella proof procedure for abductive reasoning with integrity constraints check.
2. To investigate whether partial evaluation can help to speed up the abductive reasoning process.
3. To have the gumstix computer capable of performing abductive reasoning.

**Distributed Abduction** which is to develop a suitable multi-agent system for a group of agents to collaborate in abductive reasoning tasks, and it involves the following tasks:

1. To identify the requirements of system and to investigate the possible solutions.
2. To design and implement the system
3. To identify the potential applications for the system.
4. To run the system on the gumstix computers.

1.2 Contributions

For this project, my contributions include:

1. Re-engineered an abductive meta-interpreter by Kakas’ efficiently. The new meta-interpreter runs about 43% faster and requires about 53% less memory for the Stack. The meta-interpreter can also be run on the Gumstix computers. (See Chapter 3 Investigation).

2. Investigated how partial evaluation may help to further speed-up the abductive reasoning process (for ground applications). This is done by compiling the abductive logic program with the meta-interpreter to obtain a self-abductive logic program. The initial
CHAPTER 1. INTRODUCTION

3. Designed and implemented a multi-agent system for coordinating a group of agents in collaborative reasoning tasks. The system also evolved from version 1 to version 2 along the project. (See Chapter 4 Distributed Reasoning System).

4. Proposed and implemented two distributed abductive meta-interpreters that can be integrated to the distributed reasoning system. The first meta-interpreter is based on the Kakas-Mancarella Procedure but allows agents to collaborate during reasoning. This can be considered as a general purpose distributed meta-interpreter. The second meta-interpreter is based on the abductive event calculus planner by Murray [22]. The “vanilla” planner allows several agents to collaborate during the planning process for simple reasoning tasks. The abductive meta-interpreter together with the distributed reasoning system are called Distributed Abductive REasoning System (DARES). (See Chapter 5 Distributed Meta-Interpreters).

5. Ported YAP Prolog and QuProlog for ARM Linux so that DARES can run on the Gumstix computers. This enables the Gumstix computers to behave like intelligent agents and collaborate in simple reasoning tasks. (See Chapter 6 Porting to Gumstix).

6. As “side-contributions” of the project, I helped the Gumstix community by providing the ported Prolog systems so that people can develop applications in Prolog for the Gumstix computers. I also helped the author of QuProlog to evaluate and test QuProlog version 8.0 and its new publish/subscribe server Pedro. Several bugs have been discovered, reported or fixed.

In addition to the Chapters mentioned above, Chapter 2 Background gives the knowledge you will need to understanding this project. It first gives the concept of logic programming and the “trick” of partial evaluation with logic programs. It then introduces Abductive Logic Reasoning and its current applications, in particular planning with event calculus. It also briefly describes the existing ALIAS system that coordinates abductive logic reasoning between agents. In the end, it introduces the hardware and software required for this project.

Finally, Chapter 7 Conclusion summarizes what I have done and learnt from this project. It also outlines the ideas of future development of DARES and other related research work I wish to pursue after the project.
Chapter 2

Background

2.1 Logic Programming

In 1958, John McCarthy (the father of A.I.) proposed that mathematical logic to be used for programming and invented Lisp. Since then, Lisp has become the dominated programming language for A.I in America. Later around 1972, Alain Colmerauer and Phillipe Roussel of University of Aix-Marseille, collaborated with Robert Kowalski of the University of Edinburgh to create Prolog (abbreviation for “PROGRAMming in LOGic”) as an alternative to Lisp. Now, Prolog has become the dominated programming language for A.I. in the rest of the world.

The following is a short recap of the Prolog language. First, it gives the syntax and the informal semantics for the language, then it briefly explains the algorithm Prolog uses to evaluate queries, and finally it describes an efficient implementation for most of the modern Prolog engines – Warren Abstract Machine.

2.1.1 Syntax and Informal Semantics of Prolog

Prolog Term

The single data type of Prolog is called term. A term can be a constant, a variable or a compound term.

Constant The constant terms include integers, floats and atoms:

- **Integer** can be represented in decimal (e.g. 0, 1, 123, -123), in binary (e.g. 0b1101), in octal (e.g. 0o16) and in hexadecimal (e.g. 0xff).

- **Float** such as 1.0, -3.14, 3.0e8

Atom may be in any of the following forms:

1. Any sequence of alphanumeric characters (including ‘‘), starting with a lower case letter (e.g. agent, directory_server).
2. Any sequence from the following set of characters: + - * / \ ^= < > = ? ; @ # $ &
3. Any sequence of characters surrounded by single quotes (e.g. ‘Jeffrey’, ‘artificial intelligence’).
Variable  A variable term may be any sequence of alphanumeric characters (including \$) starting with either a capital letter or \$, e.g. \texttt{X}, \texttt{Depth}, \texttt{length}. If a variable only appears once in a Prolog program or query (introduced later), we can call it \textit{anonymous variable} and use the underline character \_ to represent it.

Compound Term  Compound terms are the structured data objects of the language. A compound term has a \textit{functor} and a sequence of one or more terms called \textit{arguments}. A functor has two characteristics, the \textit{functor name} (must be an atom) and the \textit{arity} (the number of arguments of the compound term). We usually denote a functor as the form \texttt{functor name/arity}, e.g. \texttt{member/2}. A compound term has a tree structure, and its functor is the root of the tree. For example, the structural picture of the compound term \texttt{s(np(jeffrey), vp(v(likes)), np(cats))} is:

```
\[ \text{S} \rightarrow \text{np} \rightarrow \text{vp} \rightarrow \text{claire} \rightarrow \text{v} \rightarrow \text{np} \rightarrow \text{likes} \rightarrow \text{cats} \]
```

In Prolog, we can also write certain functors as \textit{operators}, e.g. 2-ary functors as infix operators and 1-ary functors as prefix or postfix operators. For example:

\begin{tabular}{|c|c|c|}
\hline
As Functors & As Operators & \\
\hline
\texttt{+(X,Y)} & \texttt{X+Y} & \texttt{infix} \\
\texttt{-(X)} & \texttt{-X} & \texttt{prefix} \\
\texttt{;(P)} & \texttt{P;} & \texttt{postfix} \\
\hline
\end{tabular}

Another important class of data structures in Prolog is \textit{List}. A list can be either the atom \texttt{[]} (or \texttt{nil}) representing the \textit{empty list} or is a compound term with functor \texttt{/2} (i.e. it has two arguments). The first argument is called the \textit{head}, whereas the second argument is called the \textit{tail} which is a list too. For example, the structural picture of a list \texttt{[a, 2, hello]} is:

```
\[ \text{a} \rightarrow \text{2} \rightarrow \text{hello} \rightarrow \text{[]} \]
```

Prolog Program and Query  
A logic program (or Prolog \textit{program}) usually describes the knowledge of a domain. The fundamental unit of a logic program is the \textit{goal}, which essentially is a Prolog term. A logic program consists of a sequence of \textit{sentences}. A sentence has a \textit{head} and a \textit{body}. The head either is a goal or is empty. The body consists of a sequence of zero or more goals.

If the head is not empty, the sentence is called a \textit{clause}. There are three types of clauses, distinguished by the number of goals in the body:
**unit clause** whose body is empty. For example:

\[ P \]

where \( P \) is the head goal and the it can be interpreted as:

Goals matching \( P \) are true.

**chain clause** whose body contains exactly one goal. For example:

\[ P : \neg Q \]

which can be interpreted as:

\( P \) is true if \( Q \) is true.

**deep rule** whose body has two or more goals. For example:

\[ P : \neg Q_1, Q_2, Q_3 \]

which can be interpreted as:

\( P \) is true if all of \( Q_1, Q_2 \) and \( Q_3 \) are true.

If the head is empty, the sentence is called a **directive**. For example:

\[ : \neg P, Q \]

can be read as:

Can \( P \) and \( Q \) be shown (or executed)?

Programs are usually defined and saved in text files known as the Prolog source files. When the programs are read by a Prolog interpreter, it will contributed to the knowledge base. A user can then submit a query about the domain the knowledge base describes to the interpreter and wait for the answer(s). A Prolog query is essentially a list of goals. For example:

\[ ?P, Q \]

which means:

Are \( P \) and \( Q \) true?

### 2.1.2 Query Evaluation in Prolog

After a Prolog program source file is read by the interpreter, the interpreter will “store” the program’s clauses in its **dynamic database**. Upon receiving a query (a conjunction of goals) from the user, the interpreter will scan the database (possibly multiple times) trying to “resolve” all of the goals and to return a set of answers if successful.
Term Unification
During the query evaluation, the most basic and important operation is to match two Prolog terms, and this process is called term unification. The process takes two terms and returns a unifier for them if one exists:

Definition 2.1 Let $S_1$ and $S_2$ be terms and $\theta$ be a substitution, if $S_1\theta = S_2\theta$ then $\theta$ is a unifier of $S_1$ and $S_2$.

In Prolog, the process of matching two terms $P$ and $Q$ can be described as below:

1. If $P$ and $Q$ are constants then $P$ and $Q$ match only if they are the same object (i.e. have the same value for integers or floats, or have the same name/value for atoms).
2. If $P$ is a variable and $Q$ is anything, then they match, and $P$ is instantiated to $Q$. Conversely, if $Q$ is a variable then $Q$ is instantiated to $P$.
3. If $P$ and $Q$ are compound terms, then they match if and only if:
   (a) $P$ and $Q$ have the same functor and arity, and
   (b) all of their arguments (subterms) match.

For example, $\text{parent}(X, \text{bob})$ and $\text{parent}(\text{jim}, Y)$ can be unified with the unifier $\{X = \text{jim}, Y = \text{bob}\}$, whereas $\text{happy}(\text{bob})$ and $\text{sad}(\text{jim})$ don’t have a unifier and they don’t match.

The Execution Black-box
The process of how Prolog evaluates a query can be illustrated as a “black-box” execution (See Figure 2.1):

Input:
- **program** A list of clauses.
- **query** A list of goals.

Output:
- **yes/no indicator** It will return yes if the black-box execution succeeds and no otherwise.
- **unifier** It will return the instantiation of variables during the execution if the execution succeeds, and an empty list if the execution fails.

The algorithm/procedure used by the black-box is called SLDNF and it can be described as below:
Execution Begin:

1. If the goal list is empty, then terminates with yes.

2. If the goal list is \([G_1, G_2, \ldots, G_n]\) and \(G_1\) is positive (i.e. it is not of the form \(\text{not } G'_1\)), then scans through the input program from top to bottom trying to find a clause \(C\) whose head can match \(G_1\):
   - If such a clause \(C\) is found and \(C\) is of the form:
     \[ H : -B_1, \ldots, B_m, \]
     then we match \(H\) and \(G_1\) (this will have the side effect of instantiating the appropriate variables in \(B_1, \ldots, B_m\)) to get \(C'\):
     \[ H' : -B'_1, \ldots, B'_m. \]
     It then starts a new black-box execution with input \([B'_1, \ldots, B'_m, G_2, \ldots, G_n]\):
     - If the new black-box execution returns yes, then terminate with yes.
     - If the new black-box execution returns no, then continue the scanning in step 2 but start from the clause defined after \(C\). This is called backtracking.
   - If no such a clause \(C\) can be found, terminate with no.

3. If the goal list is \(G_1, G_2, \ldots, G_n\) and \(G_1\) is negative (i.e. it is of the form \(\text{not } G'_1\)), then it will start a new black-box execution with input \([G'_1]\):
   - If the new black-box execution returns yes, then terminates with no.
   - If the new black-box execution returns no, then continue from step 1 with the goal list \([G_2, \ldots, G_n]\). This makes use of the concept of Negation as Failure [10].

Execution End.
If the execution terminates with yes, then the output unifier is all the unifiers obtained in step 2, either from goal and head matching or from recursive executions.
For example, consider the following Prolog program:

```prolog
edge(1,2).
edge(1,3).
edge(3,4).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

Listing 2.1: Prolog Program – Paths

Given the goal `path(1,4)`, Prolog first matched `path(1,4)` with the head of the first clause `path(X,Y) :- edge(X,Y)`, and got the new goal `edge(1,4)`. By scanning through the whole program, Prolog found out that the new goal is not provable. So it backtracked with the original goal `path(1,4)`. This time, Prolog tried to match the goal with `path(X,Y) :- edge(X,Z), path(Z,Y)` and got a list of new goals `edge(1,Z), path(Z,4)`. Prolog scanned through the program and found `edge(1,2)`, which proved `edge(1,Z)` and instantiated `path(Z,4)` as `path(2,4)`. As you can imagine, Prolog failed the subsequent steps to prove `path(2,4)`, as there isn’t an edge starting from 2. Therefore, Prolog backtracked with the goals list `edge(1,Z), path(Z,4)`. This time, Prolog found `edge(1,3)` to prove `edge(1,Z)` and got `path(3,4)` as its new goal. Now it is easy to see that Prolog would prove the final goal `path(3,4)` from the clause `path(X,Y) :- edge(X,Y)` and the fact `edge(3,4)`. The whole execution can be illustrated in an SLD-tree (See Figure 2.2).

![Figure 2.2: Query Evaluation Example (The SLD-Tree)](image)

2.1.3 Prolog Compilers and the Warren Abstract Machine

About ten years after Prolog was born, David Warren designed an abstract machine for the efficient execution of Prolog [20]. The design is known as the Warren Abstract Machine and it comprises a memory architecture and an instruction set (See Figure 2.3). WAM has now become the de facto standard target for modern Prolog compilers. In 1991, Hassan Ait-Kaci produced an excellent tutorial [2] of WAM and it has become an important documentation for implementing a Prolog engine.
Basic Components of WAM
Before explaining how WAM executes its instructions, let’s first look at its basic components. The WAM architecture consists of a set of registers and the main memory. The main memory is divided into five blocks:

**Heap** a dynamic area used for storing structural information of terms.

**Push Down List (PDL)** a work area used during unification.

**Stack** used for holding *environments* and *choice points* needed for backtracking.

**Trail** used for keeping track of the variables need to be unbound in case of backtracking.

**Code** stores the instructions for the abstract machine (obtained by compiling the Prolog programs. See Figure 2.3).

There are two types of registers:

**General Purpose Registers** are mainly used for storing structural information of a Prolog term during the unification process.

**Special Purpose Registers** are used for storing global states of the abstract machine, such as the *Heap Pointer* (H) and the *Instruction Pointer* (P).

Figure 2.3 shows the layout of the WAM’s basic components.

Heap Representation of WAM

The *Heap* can be viewed as a sequence of *cells*. There are two basic types of cells, the *Tagged Cells* and the *Functor Cells*. As its name indicates, a functor cell stores information about a functor – the functor name followed by the arity. A tagged cell contains a tag and the *datum*...
(value). There are four types of tags. A cell’s datum interpretation depends on the cell’s tag. Hence, we can say there are four types of tagged cells:

**Constant Cell** represents a constant (i.e. functor with arity 0). It is tagged with `CON` and has the constant name as its datum (numerical value for integers and floats).

**Reference Cell** represents a variable. It is tagged with `REF` and has a valid heap address as its datum. If this is an unbound variable, the datum is the heap address of the cell itself. Otherwise, it points to the heap address containing the object that the variable has been unified to.

**Structure Cell** represents a compound (structure) term. It is tagged with `STR` and has a valid heap address as its datum. The heap address points to a functor cell where the actual functor information is stored. The subterms of the compound term are stored in sequence immediately after the referenced functor cell.

**List Cell** represents a Prolog list. As explained in Section 2.1.1, a list can be represented as a pair – the head and the tail, where the tail itself is a list. A list cell is tagged with `LIS` and has a valid heap address as its datum. The datum points to the heap address at which the list’s head is stored. The list’s tail is stored in the cell immediately after
the head’s cell, e.g. another list cell. An empty list is stored as a constant, e.g. \texttt{nil} or \texttt{[]}.

For example, a possible heap representation for the term \( p(Z, h(Z, [a, b | T]), f(T)) \) is at heap address 18 (See Figure 2.5: The Heap Representation of \( p(Z, h(Z, [a, b | T]), f(T)) \)).

```
7   ...  9
8   LIS  9
9   CON  a
10  LIS  11
11  CON  b
12  REF  12
13  h/2
14  REF  14
15  REF  8
16  f/1
17  REF  12
18  p/3
19  REF  14
20  STR  13
21  STR  16
22  ...  
```

Figure 2.5: The Heap Representation of \( p(Z, h(Z, [a, b | T]), f(T)) \)

**Machine Instructions**

Prolog program containing clauses (predicate definitions) is usually saved in a text file. When a (WAM-implemented) Prolog engine “consults” the file, it will compile the clauses into a set of machine instructions that the WAM can understand and store them in the Code area. If a user enters a query, the engine will compile the query into another set of instructions and then execute it via WAM.

For example, given the predicate definition of \texttt{concat/3}:

- \texttt{concat([], L, L).}
- \texttt{concat([H|T], L, [H|R]) :-}
  - \texttt{concat(T, L, R).}

the machine instructions without optimisation could be:
concat/3:
  try_me_else   L1
  get_nil       X1  % []
  get_value     X2,X3 % L,L
  proceed
L1:
  trust_me_else_fail
  get_list      X1  % [
  unify_variable X4  % H|\n  unify_variable X1  % T1|L2,
  get_list      X3  % [
  unify_value   X4  % H|
  unify_variable X3  % T2]
execute concat/3

In the above example, the instruction try_me_else L1 means “Should this clause fails, backtrack to the clause labeled with L1”. The instruction get_nil X1 means “Is the dereferenced value of register X1 an empty list, or an unbound variable so that it can be binded to an empty list?”. The instruction get_value X2, X3 means “Try to unify the content of register X2 and that of register X3 and backtrack on failure”. Finally, proceed simply means “The execution of instructions of the current goal succeeds, so continue to execute the instructions of the next goal of the query”. For a complete description of the WAM instruction set, please refer to [2].

Structure Copying

WAM is in fact a copying implementation – it tries to build copies of structures on the heap. Let’s first consider the very basic operation – Term Unification. As we have already known, term is the basic object type of Prolog. Let’s specify two sorts of terms: a program term and a query term. Both program term and query term are first-order terms but not variables.

The idea of term unification is simple: having defined a program term p, one can submit any query q and execution either fails if p and q do not unify, or succeeds with a binding of the variables in q obtained by unifying it with p.

From the WAM’s point of view, the unification process (of two terms) is to:

1. execute the instructions for q to build an exemplar of q on the heap from q’s textual form.
2. execute the instructions for p and try to “match” p with the exemplar of q on the heap.

The matching of exemplar will behave differently according to whether it is in the read mode or in the write mode. The mode is set after executing one of the follow two instructions:

get_structure F,A
get_list A

WAM checks the argument A:

- if A is an unbound variable, the mode is set to write and the following instructions are to build a copy of the structure on the heap.
- if A is a bound, the mode is set to read and the following instructions are to check whether A is referencing to the right kind of structure and will execute the unify operation which makes use of the PDL (See Figure 2.3).
Compiling Clauses

A clause can be compiled to a set of instructions:

- the head of the clause is compiled as a program term.
- the body of the clause is compiled as a list of query terms.

However, since the same set of registers are used during the unification of each goal in the head and the body, we must protect the variables that occur in more than one body goal (they are called permanent variables). To determine whether a variable is permanent, the head atom is considered to be part of the first body goal. For example, in `p(X, Y) :- q(X, Z), r(Z, Y)`, `Y` and `Z` are permanent variables and `X` is not. Permanent variables will be saved in an environment frame (See Figure 2.4) in the stack. The creation and destruction of an environment frame are performed after the execution of the allocate and deallocate instructions. The register `E` is always pointing to the latest environment frame.

When a failure of unification (e.g., functors do not match) occurs, it should not yield irrevocable abortion of execution but considers alternative choices of clauses in the order in which they appear in definitions. Therefore, it is also necessary to save the state of computation at each procedure call offering alternatives to restore upon backtracking to this point of choice. The state of computation is stored as a choice point frame (See Figure 2.4) in the stack, and the register `B` is always pointing to the latest choice point frame.

Unbound variables may become bound during the process of unification. Therefore, it is also important to keep a list of variables which must be reset to “unbound” upon backtracking. The addresses of such variables are store in the Trail area, and the register `TR` is always pointing to the top of the trail area.

The control instructions for manipulating a choice point include `try_me_else L`, `retry_me_else L`, `trust_me` and so on. These instructions correspond to a first, an intermediate and a last clause of a predicate definition.

Optimisations

There are a number of optimisations can be considered while implementing a WAM engine (with the compiler): such as register allocation, last call optimisation, indexing (see Figure 2.6) and cut. The optimisations are confirming to a set of principles [2]:

WAM Principle 1 Heap space is to be used as sparingly as possible, as terms built on the heap turn out to be relatively persistent.

WAM Principle 2 Registers must be allocated in such a way as to avoid unnecessary data movement, and minimise code size as well.

WAM Principle 3 Particular situations that occur very often, even though correctly handled by general-case instructions, are to be accommodated by special ones if space and/or time may be saved thanks to their specificity.

2.2 Partial Evaluation in Logic Programs

2.2.1 Partial Evaluation

Partial evaluation (PE) is one type of program optimisations. It tries to transform a program’s source such that after compilation, the program will have the same semantical behaviour
but the number of computational steps needed at run-time execution is reduced. The idea was first introduced for functional programming paradigm and was then introduced in logic programming.

The basic principle of partial evaluation is to evaluate parts of the program which have enough input data and to keep the parts which don’t have enough data unchanged. The basic transformation usually has the following goals:

- unfold procedure calls. A procedure call may be substituted by its body inside another procedure.
- propagation of data structures. Data structures may be propagated as pass-in and pass-out arguments.
- evaluation of built-in predicates wherever it is possible.

For example, consider a logic program:

```
1 greater_than(X,Y) :- increment(Y,X).
2 greater_than(X,Y) :-
3     increment(Y,Z),
4     greater_than(X,Z).
5
6 increment(one, two).
7 increment(two, three).
```

Listing 2.2: Prolog Program – Ordering

If we partial evaluate the program, we can obtain the following new program:
CHAPTER 2. BACKGROUND

Listing 2.3: Example Prolog Program – Ordering (Partially Evaluated)

Hence, a query greater_than(three, X) will not cause a SLD-refutation but just the
simple unification on the predicate greater_than/2 at runtime.

There are some refinements of PE described in [25] and [3]. There are also a number of
applications of PE proposed (e.g. [19], [23]).

Next is a real example of the PE’s applications.

2.2.2 A Meta-Interpreter of Prolog

A Prolog meta interpreter can simulate the Prolog’s strategy of executing programs. A meta-
interpreter sometimes is preferred to plain execution of programs because the interpreter may
provide more information about the execution itself and may also provide more control of it.

Let’s first consider an interpreter called Demo:

Listing 2.4: Meta-Interpreter Demo

The demo interpreter takes two arguments. The first argument is a list of goals to prove,
and the second argument is the information (or explanation) of how the goals are satisfied
(proven), if demo succeeds. If we pass the logic program (see Listing 2.5) to the inter-
preter and give Prolog a query • demo([father(alan,X)],E), it will return X = bob,
E = [(father(alan,bob),true)]. This is because Prolog unifies father(alan,X) with
father(alan,bob) and father(alan,bob) is a fact in the program. Therefore, the demo
program succeeds with simply true as the explanation for father(alan,bob). Now if we
query • couple(X,Y), then the result will be X = claire, Y = alan,
E = [(couple(claire,alan),[(wife(claire,alan),true)])]. This means that
wife(claire,alan) is self-explained as it is a fact, and wife(claire,alan) also explains
couple(claire,alan).
CHAPTER 2. BACKGROUND

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- couple(X,Z), parent(Z,Y).
couple(X,Y) :- wife(X,Y).
couple(X,Y) :- husband(X,Y).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
father(alan,bob).
mother(claire,david).
wife(claire,alan).
```

Listing 2.5: Example Prolog Program – Family Tree

For a even more complicated query, we can ask `?- demo([sibling(bob,david)],E)` and the expected result will be:

```
E = [sibling(bob,david),
    [parent(claire,bob),
     [couple(claire,alan), [(wife(claire,alan),true)]],
     (parent(alan,bob),[(father(alan,bob),true))]),
     (parent(claire,david),[(mother(claire,david),true)])
]
```

2.2.3 A Partial Evaluated Meta-Interpreter

The results of the queries in Section 2.2.2 gave the full explanations to the user instead of simply answering yes. However, it involved a lot of computations as you could imagine. Surely we can apply the partial evaluation tricks to “compile” the logic program together with the demo interpreter to reduce the computation overheads. Papers proposing such a usage include [23] and [16].

For our example, one of the possible transformation would be:

\footnote{Some papers call it “compile” but it is still a source-level transformation. So please do not confuse it with the “compile” described in Section 2.1.3}
Observe the transformed program carefully, we can notice that the meta interpreter is no longer needed, and each predicate has one extra argument—this is the “explanation” argument “borrowed” from the demo interpreter. The explanation is propagated from predicate to predicate.

Now a simple query ?- father(alan, X, E) will give the same answer

\[X = \text{bob}, \ E = (\text{father(alan,bob)}, \text{true})\]

and the query ?- couple(X, Y, E) will give

\[X = \text{claire}, \ Y = \text{alan}, \ E = (\text{couple(claire,alan)}, [(\text{wife(claire,alan)}, \text{true})])\]

The more complicated query ?- sibling(bob, david, E) will still yield the correct result:

\[E = (\text{sibling(bob,david)},\]
\[\text{sibling(bob,david)},\]
\[\text{parent(claire,alan),[(wife(claire,alan),true)]},\]
\[\text{parent(alan,bob),[(father(alan,bob),true)]})\]
\[\text{parent(claire,david), [(mother(claire,david),true)]})\]

The transformed program doesn’t need the demo interpreter anymore but it still has the “demo” capacity. The transformed program also runs faster and uses less memory than that with the interpreter. This is the power of partial evaluation. Therefore, in this project we will investigate how partial evaluation can be applied to an abductive proof procedure to improve its efficiency.
2.3 Abductive Reasoning

2.3.1 Logical Reasoning

There are three methods of logical reasoning: deduction, induction and abduction.

**Definition 2.2** Given a rule \( R: \beta \leftarrow \alpha \) (read as \( \alpha \) therefore \( \beta \) or \( \beta \) if \( \alpha \)):

**Deduction** means determining \( \beta \) – using the rule \( R \) and its precondition \( \alpha \) to make a conclusion (i.e. \( \alpha \land R \Rightarrow \beta \)).

**Induction** means determining \( R \) – learning \( R \) after a large number of examples of \( \alpha \) and \( \beta \).

**Abduction** means determining \( \alpha \) – using the rule \( R \) and its postcondition \( \beta \) to assume that the precondition \( \alpha \) could explain \( \beta \) (i.e. \( \beta \land R \Rightarrow \alpha \)).

Abduction could be considered as the reverse of deduction. Here is an example, consider the following database:

\[
\begin{align*}
\text{shoes \ are \ wet} & \leftarrow \text{grass \ is \ wet} \land \text{walked \ on \ grass} \\
\text{grass \ is \ wet} & \leftarrow \text{rained \ last \ night} \\
\text{grass \ is \ wet} & \leftarrow \text{sprinkler \ was \ on}
\end{align*}
\]

If we have walked on the grass and we know that it rained last night, we can deduce that the grass is wet and our shoes will become wet. This is deductive reasoning.

However, if we observed that our shoes were wet, then we may guess that we walked on the grass and the grass was wet. We may also guess that it rained last night. The guessing of an explanation to the observations is abductive reasoning.

2.3.2 Formal Definition

An abductive framework \(< T, A >\) can be defined as follow:

**Definition 2.3** Given a set of rules \( T \) (or called theory), a goal \( O \) (or called observation), and a set of possible hypotheses \( A \) (or called abducibles), abduction is to find a set of explanation \( \Delta \) such that \( \Delta \subset A \) and \( \Delta \) satisfies:

1. \( T \cup \Delta \models O \)
2. \( T \cup \Delta \) is consistent.

Additional restrictions on explanations

In the previous example, if the set of abducibles is \{grass\_is\_wet, walked\_on\_grass, rained\_last\_night, sprinkler\_was\_on\}, then all of

\[
\begin{align*}
\Delta_1 &= \{\text{grass\_is\_wet, walked\_on\_grass, rained\_last\_night}\} \\
\Delta_2 &= \{\text{walked\_on\_grass, sprinkler\_was\_on}\} \\
\Delta_3 &= \{\text{walked\_on\_grass, rained\_last\_night}\} \\
\Delta_4 &= \{\text{walked\_on\_grass, rained\_last\_night, sprinkler\_was\_on}\}
\end{align*}
\]

are possible explanations for shoes\_are\_wet.

However, sometimes it is desired to reduce the number of candidate explanations. This can be achieved by imposing additional restrictions to the framework. Such restrictions can be:
CHAPTER 2. BACKGROUND

- **An explanation should be basic** this means an explanation should not be explained by another. For example, `grass_is_wet` in \( \Delta_1 \) can be explained by \( \Delta_3 \) (\( grass_is_wet \leftarrow \text{rained_last_night} \)). We can avoid this by restricting that no abducible predicate can appear as the head of a rule \( R \) in the theory \( T \).

- **An explanation must be minimal** this means an explanation candidate cannot be subsumed by another. For example, in the theory \( T' \)
  
  \[
  \text{grass_is_wet} \leftarrow \text{rained_last_night} \\
  \text{grass_is_wet} \leftarrow \text{rained_last_night} \land \text{sprinkler_was_on}
  \]

  , \( \{\text{rained_last_night}\} \) is a minimal explanation for \( \text{grass_is_wet} \) where \( \{\text{rained_last_night}, \text{sprinkler_was_on}\} \) is not.

- **An explanation must satisfy all of the integrity constraints** (see next).

**Integrity Constraints**

An integrity constraint is a rule used for further restricting what abducibles can be present in a candidate explanation. For example, if we add a constraint rule

\[
\leftarrow \text{rained_last_night} \land \text{sprinkler_was_on}
\]

(which can be read as *it is impossible that it rained last night and the sprinkler was on*) to \( T \), then \( \Delta_4 \) will not be a possible explanation anymore.

With integrity constraints, the definition of an abductive framework \( < T, A, I > \) now becomes:

**Definition 2.4** Given a set of rules \( T \), a goal \( O \), a set of possible hypotheses \( A \) and a set of integrity constraints \( I \), abduction is to find a set of explanation \( \Delta \) such that \( \Delta \subset A \) and:

1. \( T \cup \Delta \models O \);
2. \( T \cup \Delta \) is consistent;
3. \( T \cup \Delta \) satisfies \( I \).

### 2.3.3 Proof Procedure

**Logic Programs as Abductive Framework**

The similarity [12] between abduction and Negation as Failure [19](NAF) can be used to give an abductive interpretation of NAF. In the interpretation, negative literals are interpreted as abductive hypotheses that can be assumed to hold provided that they satisfy a set of integrity constraints together with the logic program. Hence, we can transform a general logic program \( P \) into an abductive framework (see Section 2.3.2) \( < P^*, A^*, I^* > \) where:

1. \( P^* \) is the logic program obtained from \( P \) by replacing each negative literal \( \neg p(t) \) by a new positive literal \( p^*(t) \).
2. \( A^* \) is the set of predicate symbols introduced in 1.
3. \( I^* \) is a set of all integrity constraints of the form:
   
   \[
   \forall x \neg [p(x) \land p^*(x)] \text{ and } \forall x [p(x) \lor p^*(x)]
   \]
Hence, the meaning of the set of integrity constraints $I^*$ can be defined as: for every variable-free atom $t$,

- $P^* \cup \Delta \not\models t \land t^*$
- $P^* \cup \Delta \models t$ or $P^* \cup \Delta \models t^*$

Kakas-Mancarella’s Abductive Proof Procedure

A number of abductive proof procedures and their extensions have been proposed over the years. The most influential ones are Kakas-Mancarella [17], SLDNF A [11] and IFF [15]. For this project, we will focus on the Kakas-Mancarella procedure.

It is worth noting that the original Kakas-Mancarella procedure presented was restricted such that $\Delta$ can only contain ground atoms and it doesn’t do integrity constraint checking. The proof procedure is an interleaving of two phases:

- **The Abductive Phase** which is a standard SLD-resolution – reasoning backwards for a refutation collecting any required abductive assumption.
- **The Consistency Phase** which checks for the consistency of the assumptions.

In the procedure, abducibles are divided into two sets: *base abducibles* and *non-base abducibles*. A non-base abducible is basically the negation of an original non-abducible and a base abducible is an original abducible or its negation. A refined version of the procedure can be described as follow:

$G$ is a list of ground sub-goals to be explained and has the form $\leftarrow L_1, \ldots, L_k$, where $L_i$ is either a positive or negative literal. If $k = 0$, then $G$ is $\emptyset$. The complement of $L$ is written as $L^*$. A top-down proof procedure interleaves the local abductive derivation and the local consistency derivation, to reduce $G$ to $\emptyset$ (the empty list). The $\Delta$ (a set of ground abducibles) is collected along the procedure.

**Abductive Derivation**

The abductive derivation **succeeds** if $G$ is empty. Otherwise, $G'$ is obtained by removing a literal $L$ from $G$, $\Delta'$ is the set of ground abducibles obtained after applying one of the following rules:

1. If $L$ is a non-abducible. If a rule whose head can match $L$ exists, and the instantiated body is $C$, then continue the abductive derivation on $C \cup G'$ with $\Delta' = \Delta$. Otherwise, the derivation **fails**.
2. If $L$ is an abducible and $L^*$ is already in $\Delta$, the derivation **fails**.
3. If $L$ is an abducible and $L$ is already in $\Delta$, then continue the abductive derivation on $G'$ with $\Delta' = \Delta$.
4. If $L$ is a base abducible and neither $L$ nor $L^*$ is in $\Delta$, then continue the abductive derivation on $G'$ with $\Delta' = \{L\} \cup \Delta$.
5. If $L$ is a non-base abducible and $L$ is not in $\Delta$, if there exists a consistency derivation on $\{\leftarrow L^*\}$ with $\Delta'' = \Delta \cup \{L\}$, then continue the abductive derivation on $G'$ with $\Delta'$ where $\Delta'$ is obtained after the consistency derivation. Otherwise, the derivation **fails**.
Consistency Derivation

In the consistency derivation, $F$ is a set of goals to be checked for consistency. The consistency derivation succeeds if $F$ is empty. The consistency derivation fails if $F$ contains $\emptyset$. Otherwise, let $F' \cup G = F$ and $G'$ is obtained by removing a literal $L$ from $G$ and $\Delta'$ is the set of abducibles after applying a rule. The rules in the consistency derivation are:

1. If $L$ is a non abducible. $C$ is the set of all the instantiated non empty bodies of the rules in the database whose heads can match $L$, continue the consistency derivation on $C \cup F'$ with $\Delta$.
2. If $L$ is abducible and $L$ is already in $\Delta$, continue the consistency derivation on $F' \cup G'$ with $\Delta' = \Delta$.
3. If $L$ is abducible and $L^*$ is already in $\Delta$, then continue the consistency derivation with $F'$ and $\Delta' = \Delta$.
4. If $L$ is base abducible and neither $L$ nor $L^*$ is in $\Delta$, then continue the consistency derivation with $F'$ and $\Delta' = \Delta \cup L^*$.
5. If $L$ is non-base abducible and $L$ is not in $\Delta$, if there exists an successful abductive derivation on $\{ \leftarrow L^* \}$ with $\Delta$, then continue the consistency derivation on $F'$ with $\Delta'$ where $\Delta'$ is obtained from the abductive derivation.

Now, consider an example: Let $T^*$ be

$$
\begin{align*}
 & a \leftarrow b, c. \\
 & b \leftarrow d^*, e. \\
 & d \leftarrow f. \\
 & c \leftarrow g.
\end{align*}
$$

let $A^*$ be \{e, f, g, a*, b*, c*, d*, e*, f*, g*\} and let $G$ be $\leftarrow a$.

First, it will start an abductive derivation on $G_0 \leftarrow a$ with $\Delta_0 = \emptyset$. $a$ is a non abducible, and $b, c$ is a resolvent of $a$, so $G_1 = \leftarrow b, c$ and $\Delta_1 = \Delta_0$. $b$ is a non abducible too and $d^*, e$ is the resolvent of $b$, so $G_2 = \leftarrow d^*, e, c$ and $\Delta_2 = \Delta_1$. Since $d^*$ is a non-base abducible, it puts $d^*$ to $\Delta_2$ and start a consistency derivation on $F_0 = \leftarrow d$ with $\Delta_0' = \{d^*\}$. In the consistency derivation, $d$ is a non abducible and $f$ is a resolvent to $d$, so it becomes $F_1 = \leftarrow f$ and $\Delta_1'' = \Delta_2'$. $f$ is a base abducible and both $f$ and $f^*$ are not in $\Delta_2''$, so it can put $f^*$ into $\Delta_2''$ and terminates the consistency derivation with success. Now it returns to the abductive derivation with $G_3 = \leftarrow c, e$ and $\Delta_3 = \{d^*, f^*\}$. If we carry on the procedure, it is easy to see that it will eventually reduce $G$ to $\emptyset$ and obtain $\Delta = \{d^*, f^*, e, g\}$, and the procedure will terminate with success. The computation can also be illustrated using nested boxes (See Figure 2.7).

2.3.4 Applications

Abduction is mainly used in many application categories such as fault diagnosis(e.g. medical diagnosis), high level vision, natural language understanding and planning.

Medical diagnosis and planning with abduction have been studied for a while. In medical diagnosis, the causes (diseases) may be modeled as the hypotheses and the observations are the symptoms to be explained. In planning problems, the domain knowledge is usually modeled with event calculus(See Section 2.3.1). Given a goal state at a time point, abduction is used to find a sequence of actions that will lead to the desired state at the desired time. Section 2.3.2 gives an example of planning with abduction.
2.4 Event Calculus and Abductive Planning

2.4.1 Introduction to Event Calculus

*Event Calculus* is a logical method for representing and reasoning about actions and their effects. It was first represented by Robert Kowalski and Marek Sergot in 1986, and was extended by Murray Shanahan in late 90s.

The basic components of the event calculus about actions and changes are *fluent* and *actions*. We can model the world by specifying what happens at when, and what event causes what to become true or false, etc. and then we can reason about the world’s state at a given time point. The language of the simplified event calculus has the following basic predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>holds_at(F, T)</td>
<td>Fluent F is true at time T</td>
</tr>
<tr>
<td>happens(E, T)</td>
<td>Event E happens at time T</td>
</tr>
<tr>
<td>initially(F)</td>
<td>Fluent F is true at the beginning</td>
</tr>
<tr>
<td>initiates(E, F, T)</td>
<td>Fluent F becomes true after event E happens at time T</td>
</tr>
<tr>
<td>terminates(E, F, T)</td>
<td>Fluent F becomes false after event E happens at time T</td>
</tr>
</tbody>
</table>

Here is an example of using the predicates to model a mini world $W$:
initially(have(alice, ticket)).
happens(give(alice, bob, ticket), 3).
happens(give(bob, alice, ticket), 5).
initiates(give(X, Y, W), have(Y, W), T) ← holds_at(have(X, W), T).
terminates(give(X, Y, W), have(X, W), T).

The simplified event calculus also has a set of rules $\mathcal{E}C$ for computing the state of the world:

\[
\begin{align*}
\text{holds}_\text{at}(F, T) &\leftarrow \text{initially}(F), \neg\text{broken}(F, 0, T). \\
\text{holds}_\text{at}(F, T) &\leftarrow \text{happens}(E, T_1), \text{before}(T_1, T), \\
&\quad \text{initiates}(E, F, T_1), \neg\text{broken}(F, T_1, T). \\
\text{broken}(F, T_1, T_2) &\leftarrow \text{happens}(E, T_1), \text{terminates}(E, F, T), \\
&\quad \text{before}(T_1, T), \text{before}(T, T_2).
\end{align*}
\]

Now we can reasoning about whether some fluents are true or not at a given time in our mini world. Let $\Gamma$ be a set of $\text{holds}_\text{at}s$, $\Gamma$ is true iff:

\[\mathcal{W} \cup \mathcal{E}C \models \Gamma\]

For our mini-world example, it is easy to compute that $\text{holds}_\text{at}(\text{have}(\text{alice}, \text{ticket}), 1)$ and $\text{holds}_\text{at}(\text{have}(\text{bob}, \text{ticket}), 4)$ are true but $\text{holds}_\text{at}(\text{have}(\text{bob}, \text{ticket}), 7)$ is false.

### 2.4.2 Planning with Abductive Reasoning

In the planning problems with event calculus, we can do the opposite as reasoning with event calculus.

**Definition 2.5** Let $\mathcal{E}C$ be the event calculus rules, let $\Sigma$ be a set of initiates and terminates, let $\Gamma$ be a set of $\text{holds}_\text{at}$ representing the goal state of the world. Planning with event calculus is to find a set $\Delta$ containing initially and happens such that

\[\mathcal{E}C \cup \Sigma \cup \Delta \models \Gamma\]

That is, given the effects of the events and the event calculus rules, planning is to find a set of events occur in some certain order that will lead to the desired goal state. The process of finding such set of events can be viewed as abductive reasoning. Shanahan presented an abductive event calculus planner in [22].

### 2.5 Agent Communications in Multi-Agent Systems

A multi-agent system (MAS) is a system composed of a several agents, collectively capable of reaching goals that are difficult to achieve by an individual agent or monolithic system.

MAS has been a well-established research area and lots of efforts have been put into formalising the agent communication language and protocols. Two of the well-known agent communications languages are KQML \[1\] (Knowledge Query and Manipulation Language, See Section 2.5.1) and FIPA-ACL \[14\] (Agent Communications Language standard by Foundation for Intelligent Physical Agents 3). Though KQML is superseded by FIPA-ACL, it is still used in many applications nowadays.

### 2.5.1 KQML

KQML was developed in early 1990s as part of the Knowledge Sharing Effort (insert reference). It can be used as a language for knowledge sharing between intelligent systems. It proposed a standard of message format and a set of performatives.

A KQML message has the format of:

```
(performative
 :attribute_name1 attribute_value1
 :attribute_name2 attribute_value2
 ...
 :attribute_nameN attribute_valueN
 )
```

Some of the reserved performatives \[8\] include:

<table>
<thead>
<tr>
<th>Type</th>
<th>Performative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic query</td>
<td>ask-if, ask-one, ask-all</td>
</tr>
<tr>
<td>Multi-response</td>
<td>stream-about, stream-all-eos, subscribe</td>
</tr>
<tr>
<td>Single-response</td>
<td>reply, sorry, error, tell, deny</td>
</tr>
<tr>
<td>Generic Informational</td>
<td>tell, untell, deny, achieve, unachieve</td>
</tr>
<tr>
<td>Generator</td>
<td>standby, ready, next, rest, discard</td>
</tr>
<tr>
<td>Capability</td>
<td>advertise, unadvertise, broker-one, broker-all</td>
</tr>
<tr>
<td></td>
<td>recommend-one, recommend-all, recruit-one, recruit-all</td>
</tr>
</tbody>
</table>

**Example of using the performatives for Request-response**

Below is an example of using KQML messages for request-response communication between two agent.

Alice sends a request to the directory server:

```prolog
(ask-one
 :sender alice
 :receiver directory_server
 :reply_with bob_s_number
 :language Prolog
 :ontology contact_details
 :content "work_phone(bob, Tel)"
)
```

\[3\]http://www.fipa.org/
CHAPTER 2. BACKGROUND

Directory responses with:

(ask-one
  :sender directory_server
  :receiver alice
  :in_reply_to bob_s_number
  :language Prolog
  :ontology contact_details
  :content "work_phone(bob, 12345)"
)

2.5.2 Mediation Services with KQML

KQML messages can be used in three type of mediation services — Yellow Page Directory, Mediator and Broker. The main performatives used included advertise, recommend, recruit, broker and other query and response performatives.

Yellow Page Directory Service

In the Yellow Page Directory service (Figure 2.9), there is a central directory. Each server can make advertisements on the directory. If a client needs some service, it can look up a suitable server from the central directory and then sends the requests to the “recommended” server.

Mediator Server

In the Mediator service (Figure 2.9), there is a mediator maintaining the central directory. If a client needs some service, it sends the request to the mediator. The mediator will look up the directory and forward the request to the appropriate server. The server will reply to the client directly.

Figure 2.8: Yellow Page Directory Service
2.6 Abductive LogIc Agent System – ALIAS

In [6, 24], Paolo et al. presented an agent architecture based on intelligent logic agents that are adopting abductive reasoning. The system is called ALIAS (Abductive LogIc Agent System). The system can be used for solving problems that have the following properties:
• the knowledge is incomplete
• the agents may need to find explanations about the goal domain with other agents
• the explanations have to be globally consistent over the set of agents

2.6.1 The Architecture

The system architecture coordinates several agents where each agent has its own local knowledge and can either perform abductive reasoning on their own or dynamically join other agents to cooperatively solve (explain) a goal (See Figure 2.11).

The inner structure of the each agent has two modules:

**Abductive Reasoning Module (ARM)**: This module consists of the abductive knowledge base which is represented by an abductive logic program and the abductive proof procedure\(^4\) that is used to perform abductive computation.

**Abductive Behavior Module (ABM)**: This module consists of the behavior knowledge base which is a set of LAILA clauses (see Section 2.6.2) used to define the desired actions of the agent with its environment and a LAILA interpreter for executing the behavior clauses.

In addition to the modules, each agent is maintaining a blackboard. A blackboard is used to buffer incoming LAILA queries and outgoing query results it produces. Figure 2.12 illustrates the internal architecture of an ALIAS agent.

2.6.2 The LAILA Language

LAILA \([7]\) stands for *Language for Abductive Logic Agents* and it models agent actions and interactions in a logic programming style. LAILA focuses on how agent can request other agents to prove given goals. A LAILA program is like a logic program with a set of control operators:

• \(>\) is the communication operator. For example, the query \(A_1 > G\) has the meaning of “the current agent to ask agent \(A_1\) to prove a goal \(G\)”. 

• \(↓\) is the down-reflection operator. For example, the query \(↓ G\) has the meaning of “the current agent to perform a local abduction to prove the goal \(G\)”. 

\(^4\)The proof procedure ALIAS adopts is the Kakas-Mancarella proof procedure. See Section 2.3.3.
• & is the collaborative operator. For example, the query $A_1 > G_1 \& A_2 > G_2$ has the meaning of “the current agent to ask agent $A_1$ to prove the goal $G_1$ and to ask agent $A_2$ to prove the goal $G_2$”. If both agent $A_1$ and agent $A_2$ can return an answer, the final answer is obtained by merging the returned answers into a unique consistent set of abducibles.

• ; is the competitive operator. For example, the query $A_1 > G | \downarrow G$ has the meaning of “the current agent to ask agent $A_1$ to prove the goal $G$ and the current agent to perform a local abduction on $G$”. If both answers are available, then the final answer is a non-deterministic choice between the remote answer and the local answer.

The abductive framework of LAILA has two properties:

**Local Abduction** When an agent is asked to explain a goal, it will use its local abductive knowledge base (e.g. the ARM) to perform the abductive derivation and the local consistency check.

**Global Consistency Check** An agent $X$ may ask a set of other agents to explain a goal in sequence or in parallel. When a possible explanation $\delta$ is returned, a bunch of agents $B$ that have taken part in the whole computation will also be returned. The consistency check of $\delta$ must be done individually by every single agent in $B \cup \{X\}$ via an abductive derivation in order obtain $\delta'$ which is “globally” consistent. This process could result in the expansion of $\delta$ and will continue until $\delta$ stays unchanged, or fail if one of the individual derivations fails.

Here is a simple example from [7].

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABM</td>
<td></td>
</tr>
<tr>
<td>$? A_1 &gt; q$.</td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td></td>
</tr>
<tr>
<td>$t :- \text{not} \ d.$</td>
<td></td>
</tr>
<tr>
<td>$:- \text{b, not} \ t.$</td>
<td></td>
</tr>
<tr>
<td>$q :- \downarrow r, \downarrow s.$</td>
<td></td>
</tr>
<tr>
<td>ARM</td>
<td></td>
</tr>
<tr>
<td>$r :- \text{a.}$</td>
<td></td>
</tr>
<tr>
<td>$s :- \text{b.}$</td>
<td></td>
</tr>
<tr>
<td>$:- \text{a, c.}$</td>
<td></td>
</tr>
</tbody>
</table>

The computation steps are as follow:
1. $A_0$ asked $A_1$ about $q$.

2. $A_1$ performed a local abductive derivation and local consistency check and $\delta_1 = \{a, b, \text{not } c\}$ was returned to $A_0$.

3. $A_0$ performed a local abductive derivation of $\delta_1$ and obtained $\delta_2 = \{a, b, \text{not } c, \text{not } d\}$ because of $\neg b, \text{not } t.$ and $t :\neg \text{not } d$.

4. $A_1$ was then called to perform an abductive derivation on $\delta_2$ but $\delta_2$ stayed unchanged. The whole computation terminated with the final explanation $\delta_2$.

However, it is worth noting that the final explanation obtained by the bunch of agents could be different from that obtained by a single abduction computation over the union of all agents’ knowledge bases. Consider the following example:

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABM</td>
<td>ABM</td>
</tr>
<tr>
<td>$\text{l? A_1 \succ q.}$</td>
<td>$q :\neg \downarrow r, \downarrow s.$</td>
</tr>
<tr>
<td>ARM</td>
<td>ARM</td>
</tr>
<tr>
<td>$t :\neg \text{not } a.$</td>
<td>$r :\neg a.$</td>
</tr>
<tr>
<td>$\text{:\neg b, not } t.$</td>
<td>$s :\neg b.$</td>
</tr>
<tr>
<td>$\text{t :\neg not } d.$</td>
<td>$t :\neg \text{not } d.$</td>
</tr>
<tr>
<td>$\text{:- a, c.}$</td>
<td>$\text{:- a, c.}$</td>
</tr>
</tbody>
</table>

Notice that $t :\neg \text{not } d$ in $A_0$ becomes $t :\neg \text{not } a$ and $t :\neg \text{not } d$ is now in $A_1$. The first two computation steps are still the same. However, in step 3, $A_0$ will fail the consistency check because $a$ is in $\delta_1$ but we need to prove $t$ and $\text{not } a$. However, if the ARM of $A_0$ was merged with that of $A_1$, they should have obtained same globally consistent explanation $\delta_2 = \{a, b, \text{not } c, \text{not } d\}$ because we could prove $t$ with $\text{not } d$. The main reason for ALIAS not putting all the knowledge bases together is to avoid serious computational problems.

### 2.6.3 Proposed Applications

There are two proposed applications for ALIAS.

The first one is to model judicial evaluation\[5\]. The idea is to model each character in the trial as an agent and represent the trial knowledge by means of abductive logic programming. The agents then perform collaborative/competitive reasoning on the trial data in order to obtain a consistent set of hypotheses that explains the given data.

The second application is for medical diagnosis\[4\]. In this application, the medical doctors are modeled as agents and their knowledge are represented declaratively. Given the symptoms of a patient, the doctors try work collaboratively/competitively to find the hypothesis (i.e. a disease).

### 2.7 Target Device – Gumstix

For this project, we’d like to run experimental abductive reasoning applications on the small devices. We chose gumstix computer as our target device because of its relatively low cost and open development.
2.7.1 Specification

A typical Gumstix computer consists of a mother board and some optional expansion boards. Our perspective ones will have a **connex 400xm-bt** (See Figure 2.13) motherboard, a **cfstix** (See Figure 2.14) and a **waysmall-STUART** (See Figure 2.15) expansion boards. Below is the specification of the hardware:

- Processor Speed: 400 MHz
- RAM: 64 MB
- Flash Memory: 16 MB
- Extended Storage: 1 GB (via **cfstix** with a 1 GB CompactFlash flash Card).
- Connectivity:
  - Telnet from a PC to the Gumstix – via bf waysmall-STUART and serial ports.
  - Wifi – via **cfstix** with a CF wifi card.
  - Bluetooth – via the on-board bluetooth module.
  - Ethernet (USBnet) – via the USB client.

---

The Gumstix computers are running (ARM-)Linux as their operating system. The operating system and the utility applications are compiled with ARM cross-compilers and written into a file system image in jiff2 format. The file system image can then be downloaded to the Gumstix’s flash memory via the serial port. The process is called reflashing.

Below is a list of software required for application development/porting for the Gumstix:

- HyperTerminal (Private Edition): For communications between a Windows PC and the Gumstix including Telnet and Reflashing.
- Buildroot: It is used for cross-compiling the Linux kernel and other applications. It is also used to build the file system image.
- Linux Kernel and its packages: For utilising the Gumstix hardware.

2.7.2 Resources

The Gumstix computers don’t come with any proper user guides or manuals. This makes the porting and interfacing somewhat not so straight forward. The identified resources that may help in working with Gumstix are:

- Mailing List and Email Archive [http://sourceforge.net/mailarchive/forum.php?forum=gumstix-users]: can be very useful as frequent answered questions can be searched from or posted to it.

2.8 QuProlog

QuProlog is a Prolog system developed by the University of Queensland. QuProlog is designed primarily as a prototyping and tactic language for interactive theorem provers. It provides multi-threaded supports and high-level communications between threads, process and machines. Hence, it is an ideal Prolog system for this project.
QuProlog 7.4 uses ICM (See Section 2.8.2) for peer to peer communications and uses Elvin [9] for publish/subscribe communications. However, ICM is no longer supported and Elvin package is no longer available or supported.

QuProlog 8.0 uses a new publish/subscribe server called Pedro (See Section 2.8.3) instead of ICM and Elvin.

2.8.1 Multi-threads Support

In QuProlog, one can create a new thread simply using

\[ \text{thread\_fork(ThreadName, Goal, Sizes)} \]

If \text{ThreadName} is not given, a new one will be generated automatically. \text{Sizes} is optional and is used for specifying how much memory should be allocated to this thread (e.g. stack size, code size).

A thread can retrieve its name and change its name using

\[ \text{thread\_symbol(Myname).} \]
\[ \text{and} \]
\[ \text{thread\_set\_symbol(newname).} \]

A thread can “push” a goal to another thread using

\[ \text{thread\_push\_goal(Thread, Goal).} \]

or to “kill” another thread using

\[ \text{thread\_exit(Thread).} \]

2.8.2 Inter-agent Communications Model (ICM)

This is for QuProlog 7.4 or earlier only.

The Inter-agent Communications Model (ICM) is a model of communication oriented towards the needs of inter-agent communication. It uses a store and forward architecture which allows asynchronous communication between the agents: it maintains a message buffer for each agent. The message buffer acts like a mailbox so an agent needs not to be online when a message is sent to it.

When a QuProlog process starts up, it may register with an ICM daemon (either running locally or remotely) using the switch 

\[ \text{-A processName} \]

After registered with the ICM daemon, each thread in QuProlog will have a unique handle. For example,

\[ \text{server:agent1@shell1} \]

is the thread called \text{server} within a QuProlog process called \text{agent1} running on \text{shell1} machine.

If a thread wants to send a message \text{Msg} to a thread with handle \text{B} in QuProlog, it simply does

\[ \text{Msg \texttt{->> B}} \]

If a thread wants to fetch the first message from its message buffer, it does
then \( \text{Msg} \) will be unified to the message and \( \text{W} \) will be the sender's handle. A thread can also search its message buffer for a message unifiable with \( \text{Msg} \) by \( \text{Msg} \ll\ll \text{W} \ll\ll \) and \( \ll\ll \) will suspend the thread if no such message is available.

Another powerful predicate of QuProlog is the message choice:

\[
\text{message\_choice (}
\begin{array}{l}
\text{M1 \ll\ll \text{W1} : Test1 \rightarrow \text{Call1};} \\
\text{M2 \ll\ll \text{W2} : Test2 \rightarrow \text{Call2};} \\
\ldots \\
\text{timeout(T) \rightarrow \text{CallTimeout}}
\end{array}
\)
\]

When a predicate is called, it will scan through the rules from the top. It tries to search for a message unifiable with \( \text{M} \) and satisfies the condition \( \text{Test1} \). If succeeds, it will remove the message from the buffer and call \( \text{Call1} \). Otherwise, it moves to check the next rule. If not rule is met, it will suspend the thread until a new message arrives (or timeout).

### 2.8.3 Pedro

For QuProlog 8.0 only.

Pedro is a new version (currently 0.1) of publish/subscribe server designed to replace ICM and Elvin. It supports both peer-to-peer communications and publish/subscribe communications.

Differently from ICM, all the processes/threads have to connect to the same Pedro daemon in order to have the peer-to-peer and publish/subscribe communications between them.

When a QuProlog process starts, it uses \(-A \text{procName} -N \text{pedroMachine} -P \text{pedroPort}\) to register with the Pedro server. Then the peer-to-peer messaging predicates are the same as in QuProlog 7.4.

For publish/subscribe support, any thread can publish a message \( \text{Msg} \) to the server using

\[\text{pedro\_notify(Msg).}\]

A thread needs to subscribe from the server before receiving any subscriptions. The call for subscribe is

\[\text{pedro\_subscribe(Head, Body, ID)}\]

After doing that, any future published message unifiable with \( \text{Head} \) and satisfies \( \text{Body} \) will be placed in the thread’s message buffer and the thread can retrieve it just like a peer-to-peer message (whose sender is \( \text{pedro} \)). The output \( \text{ID} \) is used for unsubscription, e.g.

\[\text{pedro\_unsubscribe(ID).}\]

Pedro also provides API for C, Java and Python. Hence, one can write applications to communicate with Pedro. This is good for writing debugger user interface.
Chapter 3
Investigation of Abductive Reasoning

The first phase of my project is to investigate how the abductive logic reasoning can be implemented efficiently or optimised so that it can be run on small devices (e.g. Gumstix).

For the investigation, I have evaluated an abductive logic reasoning meta-interpreter written by Antonis Kakas. I have also re-engineered the meta-interpreter so it runs faster than the original one (Section 3.1.1). Further more, I have tested it with a ported Prolog engine on Gumstix. For the optimisation, I have investigated how partial evaluation could help to speed up the abductive reasoning process (Section 3.2).

3.1 Abductive Meta-Interpreters

The Kakas’ meta-interpreter implements the KM proof procedure described in Section 2.3.3 but with integrity constraint checks for positive abducibles. There is a very important assumption for Kakas’ Meta-interpreter – all abducibles in the program must be ground at the time of call. This assumption helps to avoid the flundering problem (which occurs when a negative non-ground atom is put into the set of abducibles).

3.1.1 Re-engineering the Kakas’ ALP Meta-Interpreter

The Kakas’ meta-interpreter is implemented for illustrating the proof procedure. It defines its own list processing predicates (e.g. `in/2 (= member/2), concat/3) and has quite a few auxiliary predicates (e.g. empty([]).) for improving the readability of the program. However, if we want to run the the meta-interpreter on a small device, we are more interested in efficiency rather than readability. Therefore, I re-engineered Kakas’ meta-interpreter with the following modifications:

- Re-ordered the predicate arguments so that a Prolog compiler may perform “indexing” optimisation.
- Made use of `cut (!) whenever it is possible.
- Removed auxiliary predicates with empty bodies.

1The source code can be found at http://www.cs.ucy.ac.cy/aclp/
• Used available built-in predicates. Some Prolog engine has fast list processing built-in predicates. So we should avoid re-defining those predicates ourselves.

• Re-ordered predicate definitions and merge them wherever it is possible. For example, instead of testing the literal type (e.g. base or non-base abducible, or non abducible) in all the definitions of a predicate (e.g. `demo_one`), we can test it once and call the appropriate definition.

The re-engineered version of the meta-interpreter is listed as Listing 3.1. Please see Appendix A.1 for the Kakas’ meta-interpreter.

As you can observe from Kakas’ meta-interpreter, all of `demo_one/3` need to test the type of the first argument. If we call `demo_one/3` with a positive ground abducible which is not assumed, the 6th rule (line 78) will be selected after calling `is_negative/1` three times and `is_positive/1` two times and `abducible/1` two times (i.e. failing the first 5 rules). This is because it doesn’t make use of the indexing optimisation by the WAM compiler (See Figure 2.6). In my meta-interpreter, I use `abdemo_one/3` to test the type of the given literal once, and then “dispatch” it to the appropriate `abdemo_one/4`. With the magic of WAM compiler, the `abdemo_one/4` clauses will be indexed. When a positive abducible is given, the `abdemo_one/4` in line 21 will be “called” immediately. This removes the overhead of testing the literal type again and again.

In the Kakas’ meta-interpreter, the predicates such as `select_first_denial/3`, `select_literal/3` and `empty([])` are just for improving readability. In my meta-interpreter, all such “dummy” predicates are removed. In addition, the Kakas’ meta-interpreter defines its own list processing predicates such as `in/2`, `concat/3` and `select/3`. In some of the modern Prolog systems, these predicate are built-in with efficient implementation (not necessary declarative). In my meta-interpreter, I use the built-in predicate (such as `member/2`, `append/3`) wherever it is possible.

There are some other changes such as splitting the `demo_one/3` into more smaller rules to improve the flow of logic. In a word, the Kakas’ meta-interpreter is written declaratively and is suitable for illustrating the proof procedure, where my meta-interpreter is written procedurally and is suitable for running real applications (for small devices). See Section 3.1.2.

It is worth noticing that the `abdemo_naf_one`, `abdemo_naf` and `abdemo_naf_all` in my meta-interpreter are corresponding to the `demo_failure_on_literal`, `demo_failure_leaf` and `demo_failure` in Kakas’ meta-interpreter respectively.

```prolog
1 :- use_module(library(lists)).
2 %:- dynamic a/1, ic/1, rule/2.
3
4 % Abductive Derivation Phase with IC check on positive abducibles
5 abdemo([], D, D).
6 abdemo([G|Rest], Di, Do) :-
7     abdemo_one(G, Di, D1),
8     abdemo(Rest, Di, D1).
9  
10 abdemo_one(G, Di, Do) :-
11     literal_type(G, T), !,
12     abdemo_one(T, G, Di, Do).
13
14 abdemo_one(neg_ab, G, D, D) :-
15     member(G, D), !.
```
abdemo_one (neg_ab, not(P), D, _) :-
  member(P, D), !, fail.
abdemo_one (neg_ab, G, D, [G|D]).

abdemo_one (pos_ab, G, D, D) :-
  member(G, D), !.
abdemo_one (pos_ab, G, D, _) :-
  member(not(G), D), !, fail.
abdemo_one (pos_ab, G, Di, Do) :-
  ic_check(G, [G|Di], Do).

abdemo_one (non_ab, G, Di, Do) :-
  rule(G, Gs),
  abdemo(Gs, Di, Do).

abdemo_one (non_base_ab, G, D, D) :-
  member(G, D), !.
abdemo_one (non_base_ab, not(P), Di, Do) :-
  abdemo_naf([P], [not(P)|Di], Do).

% Consistency Derivation

abdemo_naf_one (G, Di, Do) :-
  literal_type(G, T), !,
  abdemo_naf_one (T, G, Di, Do).

abdemo_naf_one (neg_ab, not(P), D, D) :-
  member(P, D), !.
abdemo_naf_one (neg_ab, G, Di, Do) :-
  member(G, Di), !, fail.
abdemo_naf_one (neg_ab, not(P), Di, Do) :-
  abdemo([P], Di, Do).

abdemo_naf_one (non_base_ab, G, Di, Do) :-
  member(G, Di), !, fail.
abdemo_naf_one (non_base_ab, not(P), Di, Do) :-
  abdemo([P], Di, Do).

abdemo_naf_one (pos_ab, G, D, D) :-
  member(not(G), D), !.
abdemo_naf_one (pos_ab, G, Di, Do) :-
  member(G, Di), !, fail.
abdemo_naf_one (pos_ab, G, D, [not(G)|D]).

abdemo_naf_one (non_ab, G, Di, Do) :-
  findall(Gs, rule(G, Gs), Gss),
  abdemo_naf_all(Gss, Di, Do).

ic_check(G, Di, Do) :-
  findall(Gs, (ic(IC), member(G, IC), delete(IC, G, Gs)), Gss),
  abdemo_naf_all(Gss, Di, Do).

abdemo_naf_all([], D, D).
abdemo_naf_all([Gs|Rest], Di, Do) :-
CHAPTER 3. INVESTIGATION OF ABDUCTIVE REASONING

3.1.2 Benchmarking

For benchmarking between the (my) re-engineered meta-interpreter and the original one (Kakas'), I wrote a program to generate very big stratified logic programs. For example, one of the test logic program has 20000 abducibles and 20000 rules:

```
% all bxxx(X) and cxxx(X) are abducibles
% all axxx(X) are non abducibles. each axxx(X) has % two bodies.
% there is no need for integrity constraints as % they can be translated into a set of corresponding % clauses.
abducible_predicate(b1).
abducible_predicate(c1).
  % 20000 lines omitted
abducible_predicate(b10002).
abducible_predicate(c10002).

a0(X) :- b2(X), a1(X).
a0(X) :- c2(X), not(a1(X)).
a1(X) :- b3(X), a2(X).
a1(X) :- c2(X), not(a3(X)).
a2(X) :- b3(X), a3(X).
a2(X) :- c3(X), not(a3(X)).
  % 40000 lines omitted
a9999(X) :- b10001(X), a10000(X).
a9999(X) :- c10001(X), not(a10001(X)).
```

I didn’t generate the integrity constraints for the test programs because it is hard to check whether the randomly generated integrity constraints can keep the original program...
CHAPTER 3. INVESTIGATION OF ABDUCTIVE REASONING

stratified. If the test program is not stratified, then both of the two meta-interpreters may not terminate (because SLD is not complete). In fact, the process of integrity constraint check is similar to the process of failing a non-abducible (e.g. it needs to fail all the resolvents). Therefore, using the above generated test programs will not lose generality.

I tested the two meta-interpreters with a number of queries. They were run with YAP Prolog on an Linux machine with Pentium(R) 4 HT 3.20GHz processor and 1GB memory. They gave the same answers and a segment of the performance comparison table is shown below:

<table>
<thead>
<tr>
<th>Queries</th>
<th>Re-engineered Version</th>
<th>Kakas’ Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU Time (seconds)</td>
<td>Max. Stack (bytes)</td>
</tr>
<tr>
<td>a0(test)</td>
<td>3.163</td>
<td>1073152</td>
</tr>
<tr>
<td>a1(test)</td>
<td>2.781</td>
<td>1073152</td>
</tr>
<tr>
<td>a2(test)</td>
<td>1.876</td>
<td>1073152</td>
</tr>
</tbody>
</table>

Other queries and random generated logic programs were also tested. On average, the re-engineered meta-interpreter runs 43.11% faster and requires 53.06% less memory.

3.1.3 Running On a Gumstix

The re-engineered meta-interpreter was also run on a Gumstix with the ported YAP Prolog engine (See Chapter 6). For testing and benchmarking results, please see Section 6.2.

3.2 Optimisation by Partial Evaluation

In Section 2.2 there is an example showing how partial evaluation may “compile away” the deductive meta-interpreter demo/2 (See Listing 2.4) and improve the runtime performance. By observation, we know that there is a similarity between demo/2 and the abductive meta-interpreters: in demo/2, the justification of each successful goal is collected and returned at the end. In the abductive meta-interpreter, the list of abducibles (Δ) is collected and returned at the end. However, the difference is, the justification collected by demo/2 doesn’t affect the execution of demo, but the Δ for the abductive meta-interpreters does – e.g. it needs to be checked for consistency before adding any new abducible.

In this project, I have investigated the idea of “compiling away” the abductive meta-interpreter in attempt to improve the runtime performance of the abductive reasoning procedure. The investigation result shows that partial evaluation does help to speed up the reasoning process and requires smaller stack size. However, the trade off is that the code size of a transformed program is much bigger than the original one. I have also written a partial evaluator for compiling away the abductive interpreter.

3.2.1 Observations of the Abductive Meta-interpreter

Let’s look at the re-engineered version of the meta-interpreter(See Listing 3.1) carefully. There are some interesting observations.

Observations for base abducibles

Base abducibles are literals with abducible predicates. By comparing the abdemo_one/4 for negative base abducibles (line15 ~ line19) and the abdemo_naf_one/4 for positive base
abducibles \((\text{line 55} \sim \text{line 59})\), we can see that they have one to one corresponding identical definitions. This is the same for \texttt{abdemo\_one/4} for positive base abducibles \((\text{line 21} \sim \text{line 26})\) and the \texttt{abdemo\_naf\_one/4} for the negative abducibles \((\text{line 43} \sim \text{line 48})\). Their first two definitions (for \(\Delta\) check) are the identical. In addition, the definition

\begin{verbatim}
abdemo\_naf\_one(neg\_ab, not(P), Di, Do) :-
    abdemo([P], Di, Do).
\end{verbatim}

first calls \texttt{abdemo/3}, and then will call

\begin{verbatim}
abdemo\_one(pos\_ab, G, Di, Do) :-
    ic\_check(G, [G|Di], Do).
\end{verbatim}

So they are the same again.

According to our observation, we know that the abductive derivation phase and the consistency derivation phase for base abducibles are symmetric, therefore, we should be able to merge the 12 rules into 6. Further more, the delta checks are the same too – i.e. if the complement of the base abducible is in \(\Delta\) then the \texttt{abdemo/4} fails; if the base abducible is already in \(\Delta\), then the \texttt{abdemo/4} succeeds. Hence, we should also be able to merge the the 4 check delta rules into 2. In conclusion, we only need 4 \texttt{abdemo\_one} rules for both the positive and negative in both the abductive and consistency derivations.

**Observations for non base abducibles and non abducibles**

Non base abducibles are negative literals with non-abducible predicates, where non abducibles are positive literals with non-abducible predicates. Let’s look at the \texttt{abdemo\_one/4} (\texttt{line 28}) and \texttt{abdemo\_naf\_one/4} (\texttt{line 61}) for non abducibles. The \texttt{abdemo\_one/4} succeeds if it can succeed in an abductive derivation for at least one but any one resolvent of the current non-abducible, where the \texttt{abdemo\_naf\_one/4} succeeds only if it can fail every consistency derivation for all the resolvent of the current non abducible.

Let’s look at the case for non-base abducible. There are two rules for \texttt{abdemo\_one/4} and \texttt{abdemo\_naf\_one/4} respectively. The first rules of them are symmetric. But second rules seem to be symmetric by not exactly – i.e. the one of \texttt{abdemo\_naf\_one/4} adds the non-base abducible to \(\Delta\) where the other one doesn’t.

In conclusion, we need 6 rules for the the non-base abducibles and non abducibles in the abductive and consistency derivations.

### 3.2.2 Compiling Away the Meta-Interpreter

After the observation, we can try to transform a logic program into a hybrid program that on its own can perform abductive reasoning. I will explain this process with an example. Below is a normal logic program for diagnosis of a car.
/* rules */
rule(car_doesnt_start(X), [battery_flat(X)]).
rule(car_doesnt_start(X), [has_no_fuel(X)]).
rule(lights_go_on(X), []).
rule(fuel_indicator_empty(X), []).

/* integrity constraints */
ic([battery_flat(X), lights_go_on(X)]).
ic([has_no_fuel(X), not(fuel_indicator_empty(X)), not(broken_indicator(X))]).

/* abducibles */
ab(battery_flat(X)).
ab(has_no_fuel(X)).
ab(broken_indicator(X)).

Listing 3.2: Car Example (To Be Partial Evaluated)

Generating rules for abducible predicates

First of all, I defined a new predicate member2/3:

% member/3 takes three arguments.
% it expects a (ground) base abducible as the
% first argument and the current delta as the
% second argument.
% member/3 succeeds if and only if the base
% abducible is in delta (+) or its complement
% is in delta (-).
member2(A, [not(A)|_], -) :- !.
member2(A, [A|_], +) :- !.
member2(A, [_|B], C) :-
    member2(A, B, C).

Now let’s generate the rules for abducible predicates. According to our observation in Section 3.2.1, we only need 4 rules for an abducible predicate.

For an abducible without any integrity constraint, like the broken_indicator(X), we can transform it to be

broken_indicator(Sign, Delta, Delta, X) :-
    member2(broken_indicator(X), DeltaIn, A), !,
    A == Sign. % it is in the delta, not its complement
broken_indicator(+, DeltaIn, [broken_indicator(X)|DeltaIn], X).
broken_indicator(-, DeltaIn, [not(broken_indicator(X))|DeltaIn], X).
The first argument indicates whether this rule is for a positive or for a negative base abducible. The second argument is the input delta and the third argument is the output delta. The fourth argument is the argument of \texttt{\texttt{broken_indicator}(X)}\texttt{, i.e. X}. The first rule is actually combining the check delta rules for positive and negative base abducibles. The second rule is for a positive base abducible in the abductive derivation or a negative base abducible in the consistency derivation. The third rule is the opposite of the second rule.

For an abducible with integrity constraint, like the \texttt{\texttt{has\_no\_fuel}}, we can transform it to be

\begin{verbatim}
has_no_fuel(Sign, Delta, Delta, X) :-
in2(has_no_fuel(X), DeltaIn, A), !,
   A == Sign.
has_no_fuel(+, DeltaIn, DeltaOut, X) :-
   (fuel_indicator_empty(-, fail,  
     [has_no_fuel(X)|DeltaIn], DeltaOut, X);
   broken_indicator(+,
     [has_no_fuel(X)|DeltaIn], DeltaOut, X)).
has_no_fuel(-, DeltaIn, 
   [not(has_no_fuel(X))|DeltaIn], X).
\end{verbatim}

In this example, only the format of the second rule is different from that for the previous predicate. \texttt{\texttt{has\_no\_fuel}/1} has an integrity constraint such that it cannot holds if \texttt{not(fuel\_indicator\_empty(X))} and \texttt{not(broken\_indicator(X))} both hold. Hence, for the second rule, it succeeds if we can fail to prove \texttt{not(fuel\_indicator\_empty(X))} or fail to prove \texttt{not(fuel\_indicator\_empty(X))}.

In the body, the predicate

\begin{verbatim}
fuel_indicator_empty(-, fail, [has_no_fuel(X)|DeltaIn], DeltaOut, X)
\end{verbatim}

The - sign indicates this is a negative literal, \texttt{fail} means fail to prove it (i.e. Negation As Failure).

Since failing the negation of an abducible is the same as succeeding in abducing it, so the call to

\begin{verbatim}
broken_indicator(+, [has_no_fuel(X)|DeltaIn], DeltaOut, X)
\end{verbatim}

is correct.

\textbf{Generating rules for non-abducible predicates}

For a non-abducible predicate, like \texttt{\texttt{car\_doesnt\_start}(X)}, we can transform it to
car_doesnt_start(-, true, Delta, Delta, X) :-
    member(not(car_doesnt_start(X)), Delta), !.
car_doesnt_start(-, true, DeltaIn, DeltaOut, X) :-
    car_doesnt_start(+, fail, 
                   [not(car_doesnt_start(X))|DeltaIn], DeltaOut, X).
car_doesnt_start(-, fail, DeltaIn, DeltaOut, X) :-
    member(not(car_doesnt_start(X)), DeltaIn), !,
    fail.
car_doesnt_start(-, fail, DeltaIn, DeltaOut, X) :-
    car_doesnt_start(+, true, DeltaIn, DeltaOut, X).
car_doesnt_start(+, true, DeltaIn, DeltaOut, X) :-
    ( battery_flat(+, DeltaIn, DeltaOut, X) ;
      has_no_fuel(+, DeltaIn, DeltaOut, X) )
    .
car_doesnt_start(+, fail, DeltaIn, DeltaOut, X) :-
    battery_flat(-, DeltaIn, D1, X),
    has_no_fuel(-, D1, DeltaOut, X).

In this example, the first argument of car_doesnt_start/5 is the sign and the second argument is the type of derivation. For example, if we want to call abdemo_one for a non-abducible, the sign will be + (this is a positive literal) and the second argument will be true, which means to succeed in resolving it. The above 6 rules are corresponding to the 6 rules for literal with non-abducible predicates we observed in the Section 3.2.1.

Example Run

Other predicates in the program are transformed similarly. Now instead of typing:

    abdemo([car_doesnt_start(yourcar)], [], D)

We just simply use

    car_doesnt_start(+, true, [], D, yourcar)

instead, and the answers returned by the meta-interpreter and the transformed program is exactly the same:

D = [has_no_fuel(yourcar)]
D = [broken_indicator(yourcar), has_no_fuel(yourcar)]

3.2.3 Benchmarking

I have written a partial evaluator peval for transforming logic programs in the way I explained. I have also done a simple benchmarking for it.

For testing, I transformed the generated huge stratified program describe in Section 3.1.2 and ran the transformed program. Below is the result of the test runs of the transformed program:
CHAPTER 3. INVESTIGATION OF ABDUCTIVE REASONING

<table>
<thead>
<tr>
<th>Queries</th>
<th>Meta-Interpreter</th>
<th>Transformed Program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU Time (seconds)</td>
<td>Max. Stack (bytes)</td>
</tr>
<tr>
<td>a0(test)</td>
<td>3.163</td>
<td>1073152</td>
</tr>
<tr>
<td>a1(test)</td>
<td>2.781</td>
<td>1073152</td>
</tr>
<tr>
<td>a2(test)</td>
<td>1.876</td>
<td>1073152</td>
</tr>
</tbody>
</table>

On average, the transformed program is about 56.3% faster than the re-engineered version of the meta-interpreter and requires about 57.8% less stack space. However, as you may have guessed, the transformed program is much bigger than the original one. This is because for each predicate, it generates 4 to 6 rules. This results in the huge need in memory for the code block. As for the statistics, the meta-interpreter plus the original test program required 9457664 bytes (≈ 9 MB) memory for the code block, where the transformed program required 35454972 bytes (≈ 35 MB).

3.2.4 Conclusion

My experimental results show that partial evaluation can help to further optimise the meta-interpreter (in fact, it is to compile the meta-interpreter with the original logic program):

- Partial evaluation tests the literal type (abducible or non-abducible, etc.) at compile time where the meta-interpreter to needs to check it again and again during runtime.

- Partial evaluation performs the finding of related integrity constraints for abducibles and the finding of resolvents for non-abducibles at compile time, where the meta-interpreter does those during runtime using the predicate `findall/3`.

However, though the speed has been improved, the extra space required for storing the transformed program is a big down effect and hence it is not particularly suitable for small devices.
Chapter 4

Distributed Reasoning System

As for the second goal of this project, I have designed and implemented a system that allows a group of agents to perform collaborative reasoning tasks. The system allows the loosely coupled agents to discover and ask suitable helpers during the reasoning process. The system also provides an efficient way for agent interaction. This chapter presents the design of the system.

4.1 Overview

4.1.1 Motivation

In some simple reasoning system, there is only one agent. The agent is responsible for both maintaining the single knowledge base and performing reasoning on the domain its knowledge describes. However, the “God” agent approach implies that the single reasoning agent must:

1. know almost everything about the domain and the sub-domains.
2. do all the reasoning on its own.

In practice, this is not ideal because it is often that the knowledge base is incomplete and/or the agent’s computational power is limited. To be more sufficiently intelligent, a reasoning system may be engineered as a collection of many loosely coupled but cooperative reasoning agents.

For example, in a diagnosis system comprises a number of “expert” agents. Each agent has its own expertise about a sub-domain but doesn’t want to share all of its knowledge with others. When a fault in the target system is observed, the experts will need to collaborate together to find a possible explanation to the fault. During the collaboration, when an expert couldn’t explain a “goal” with only its own knowledge, some other agents may help to explain it according to their expertise. When an expert raises an explanation, it also needs to check the explanation with all other experts in order to make sure the explanation is consistent (agreed by every expert).

Here is another situation where there is a collection of robots working distributively within an environment. Each robot may have some certain perceptions and may perform some certain actions. Therefore, each robot is maintaining its own knowledge about the region it is currently in. Given a goal state of the environment, the robots may need to
collaborate together to find a consistent plan (sequence of actions) for it. Alternatively, the collection of robots may use their perceptions to reasoning about the environment.

Sometimes it is possible to re-engineer the multi-agent systems in the previous examples to be the single-agent systems, i.e. merge together all the agents’ knowledge base and perform a single abductive reasoning or planning from the “global” knowledge base. However, using the multi-agent approach will have the following major advantages:

- It makes use of parallelism. For example, a goal can be sent to several agents so that they can try to solve it simultaneously. In the situation where the agent knowledge bases are big and disjunctive, this advantage is significant comparing to a single multi-thread agent with a single knowledge base.

- It makes hierarchical planning more efficient. For example, we can have a coordinate robot to find a top level plan comprises decomposable goals and then send the goals to different robots to solve.

In the cases where the knowledge base is not mergeable, the multi-agent approach is unavoidable and hence distributed reasoning is needed.

4.1.2 Limitations of ALIAS

ALIAS, as introduced in Section 2.6, is a system for coordinating the collaboration between a set of abductive reasoning agents. However, there are some limitations of it:

- The interactions between the agents are pre-defined by a set of “rules” written with the LAILA language. Therefore, when an agent fails to prove a goal alone, it cannot ask for help from other agents if the LAILA rules do not instruct it to do so.

- It assumes that each agent knows about its helpers all the time. A new agent cannot contribute to a reasoning task until the behavioral knowledge bases of some of the existing agents are changed so that the new agent’s existence is aware by them.

- It only works for abductive reasoning agents.

Hence, the ALIAS system is not what exactly we are looking for. However, it does give some valuable ideas about how to coordinate a set of abductive reasoning agents during their collaborations. These ideas have helped me to design a more generic and flexible system for coordinating multi-agent reasoning.

4.1.3 Basic Assumptions

The system should be generic enough for a collection of agents to perform various reasoning tasks. “generic” means that the system is independent from how the global knowledge base is distributed across the agents, and what type of reasoning (for this project I focused on the deductive and abductive reasoning only) the agents are performing.

Assumption 4.1 The knowledge base of the agents in the system can be overlapped or disjoint. Two knowledge bases are disjoint if they do not share the the same definition for a predicate.

This (version of the) system is designed for coordinating collaborative reasoning. Therefore, it is not designed for handling various network attacks or fatal network failures. Also, it is assumed that each agent is rational and can be trusted by each other.
Assumption 4.2 *The communication channel is safe and reliable.*

In the above assumption, “Reliable” means the message sent between two agents will not be lost or corrupted. This may be achieve by implementing the communications using TCP.

Assumption 4.3 *Each agent is rational and is trusted by others.*

This prevents a malicious agent from interfering the collaborations between other agents.

### 4.1.4 Desired Properties

Before specifying the expected properties of the system, we should first understand the following terminology:

**Goal**: A goal is a list of predicates to be proven or solved. In planning, a goal can be a list of final states we would like to achieve.

**Result/Answer**: A result or an answer is associated with a goal. For example, in abductive reasoning, a result is a list of abducibles that together with the knowledge base can entail the associated goal. Where in a planning example, a result is a possible plan that if it is executed by the agents, it will lead to the expected final states (i.e. the goal).

**Agent Group** refers to the total collection of the agents currently in the system.

**Agent Cluster** refers to the set of agents currently collaborating to prove a global goal. For example, if agent 1 asks agent 2 to help with proving a sub-goal while it is trying to prove the top level goal, then agent 1 and agent 2 form a cluster. The definition of agent cluster is important when we talk about *global consistency* of a result (e.g. is it consistent among the agent group or consistent among the agent cluster?).

Now we can look at the desired properties of such a distributed reasoning system.

First, there isn’t a fixed number of agents in a distributed system and the agents are loosely coupled. This means that an agent can join or leave the group at any time providing that it follows some pre-specified protocols so that its arrival or departure will not destroy the integrity of the system. For example, a correct agent should always notify the group on its arrival and departure.

**Property 4.1** *An agent can join or leave the system at anytime by following a safe protocol that guarantees the integrity of the system.*

The agents are loosely coupled also means that the agents may or may not know who are in the group or what reasoning capability another agent has. Therefore, a way of agent/service discovery is needed.

**Property 4.2** *The system should allow an agent to discover other agents in the group and query about other agents’ capabilities.*

Hence, an agent can identify and ask potential helpers while trying to prove a subgoal. However, depending on the type of reasoning tasks, a result returned by a helper must be checked for consistency. For example, in an abductive reasoning task, one should make sure the returned result (a set of abducibles) does not violated the integrity constraints of all the agents in the cluster. Otherwise, the returned result is considered to be no good, and the agent should ask for further results from the helper or continue backtracking.
Property 4.3 A final result obtained for a top level goal must be “agreed” by all the agents in the cluster, i.e. it is consistent within the cluster.

During some reasoning processes, two agents in a cluster may ask each other for help about different sub-goals. For example, agent 1 resolves a top level goal \( G \) into a list of sub-goals \( g_1, \ldots, g_k, \ldots, g_n \), and then asks agent 2 about \( g_k \) at some point. But agent 2 may resolve \( g_k \) into another list of sub-goals \( g'_1, \ldots, g'_i, \ldots, g'_m \) and asks agent 1 about \( g'_i \). In this case, agent 1 will have two different reasoning tasks at the same time. In fact, this can be viewed as that agent 1 is given two different goals and will process them in parallel.

Property 4.4 An agent can perform different reasoning tasks at the same time, which implies that it may belong to different clusters at the same time.

At the first attempt, I designed and implemented the first version of the system. As the design evolved, I then implemented a second version of the system attempting to improve the system’s performance. The main difference between the two versions of the system is the mechanism used for agent discovery. They have their own advantages depending on the nature of the applications. In the following sections, I will first describe the two version of the system in details, and then give a comparison between the two, and finally present an example of collaborative reasoning using the system.

4.2 System Version 1

This is the very first and experimental design of the system. The design aims to satisfy all of the properties describe in Section 4.1.4. In particular, the system adopts the Yellow Page Directory (Section 2.5.2) approach to satisfy Property 4.2 and a protocol is developed for Property 4.1. Each agent is multi-threaded so that they can process different requests at the same time and hence satisfy Property 4.4. Each agent can handle different reasoning requests by using the corresponding Meta-Interpreters. For Property 4.3, since the criteria of consistency depends on the type of the reasoning tasks, the property will be satisfied by the design of the meta interpreters.

This version of the system has been implemented with QuProlog 7.4.

4.2.1 Architecture Overview

There are two basic types of agents in the system – a Directory Server Agent (Server or DA for short) and a collection of Reasoning Agent (Agent or RA for short). An agent informs the server about its reasoning capability by sending advertisements to the server. The server is responsible for maintaining such advertisements and keeps a record of each agent’s current status (e.g. is it active?). An agent can perform either local reasoning or global reasoning on a given goal. While reasoning locally, the agent will only try to prove/solve the goal using its own knowledge and needs to make sure the result is consistent with its integrity constraints. While reasoning globally, the agent will first try to prove/solve a goal by itself just like performing a local reasoning. If it fails to do so, it can ask other agents to help with one or more of the sub-goals. As the agent doesn’t know about other agent’s reasoning capability, it will first query the server for a list of potential helpers (the agents have advertised to solve the given goal), and then ask some or all of the helpers to solve the goal in parallel. As soon as an answer is returned from any of the helpers, the agent can use it to continue its reasoning. Should the reasoning fails, the agent can backtrack and ask for the next answer from the helpers. For distributed reasoning, an answer is associated with its original goal as...
well as an agent cluster. If a helper is asked to solve a goal and succeeds in doing so, it (and whoever it also asks) will be added to the agent cluster where the request agent is in. Either the helper or the request agent must make sure the returned answer is consistent within the associated agent cluster before it is used for the rest of the reasoning process.

Figure 4.1 shows an overview of the relationship between the server and agents and the relationship between the agents and clusters. Initially, each agent doesn’t know each other and tries to perform reasoning on its own (by default). E.g. RA9 alone forms a cluster. Whenever an answer is returned from RA9, it is associated with cluster 4.

When an agent fails to solve a goal on its own, it may query the server and ask for help from potential helpers. E.g. RA1 queries the server and discovers RA2 and RA3. It then asks them simultaneously. RA3 can solve the given sub-goal alone, but it needs to check with RA1 to make sure that the answer it produces is consistent between them. So RA1 and RA3 form a cluster. On the other hand, RA2 cannot solve the given sub-goal itself. However, it may break the sub-goal into another list of smaller sub-goals and ask other agents to help to solve them all. In this example, RA2 queries the server and discovers RA4, and they can indeed solve the RA1’s sub-goal together. Similarly, they need to check their answer with RA1 for consistency. Hence, RA1, RA2 and RA4 form another cluster. Finally, RA1 will have two different answers for its top level goal – one is associated with cluster 1 and the other is associated with cluster 2.

There is another case in the example: RA5 asks RA6 to prove a sub-goal and receives an answer. At that moment, cluster 3 contains RA5 and RA6. Then RA5 continues the reasoning process with the received answer. Later it gets stuck again and asks RA7 to solve another sub-goal. RA7 solves that with RA8. Now cluster 3 expands and contains all of the four agents, and the answer returned from RA7 to RA5 must be checked for consistency.
among the agents in cluster 3. If at the end RA5 solve all the rest of the sub-goals without asking any more new agents, the final answer produced by RA5 will also be associated with cluster 3 – they have participated in the reasoning process so the final conclusion (answer) must be agreed by all of them.

4.2.2 Inter-agent Communication

Two agents must be able to talk to each other in order to collaborate in a reasoning process. A good communication protocol can avoid chaos during agent collaboration. For example, if an agent asks a large number of helpers simultaneously, and all the helpers try to send all the answers they produced to the agent roughly at the same time, then the communication channel is likely to be jammed. Also for example, an agent can ask for help at some point and ask for help again later during a reasoning process, how can the agent handle the received answers correctly during backtracking?

In this section, I will focus on the high-level aspect (i.e. message format, message order) and the low-level aspect (i.e. how a message is delivered and buffered) of the agent communication. In Section 4.2.4 I will talk about how an agent handles incoming and outgoing messages.

Low Level Communication

This version of system makes use of ICM(See Section 2.8.2) for inter-agent communication (as well as inter-thread communication within an agent). Each agent (strictly speaking, “each thread” within an agent. See Section 4.2.4) has its own message (buffer) queue maintained by a ICM daemon on the local host. When an agent wants to send a message to a target agent, it will pass the message and the target agent’s handle to the local ICM daemon. The local ICM daemon will then sends the message with the source agent’s and target agent’s handles to the ICM daemon on the host machine where the target agent is running on via a TCP connection. Upon receiving the message, the target ICM daemon will associate the message with the source agent’s handle and place it to the target agent’s message queue. The target agent can retrieve the message anytime it wants. Figure 4.2 illustrates the process.

![Figure 4.2: Agent Communication via ICM](image)

The communications between agents on the same host and the communications between threads of an agent process is the same except that the source ICM daemon is the target ICM daemon (i.e. no network communication needed).

When an agent’s message queue is maintained by an ICM daemon, the agent can manipulate the queue in three ways:
• Checks the first message in the queue without taking it off the queue.
• Takes the first message off the queue.
• Searches for a message in the queue and takes it off if found.

The consequence of using ICM is that all the communications are in fact asynchronised. E.g. although an agent has sent a message to a target agent’s buffer successfully, it doesn’t know whether or not the target agent has read the message. In order to achieve synchronised message passing, a higher level communication protocol is needed. For example, after sends out a message, the source agent can suspends and wait for an acknowledgement message from the target agent.

High Level Communication

The high level communication requires a fixed message format so that the agents can parse them correctly. Also, the messages should be categorised and sent in a pre-agreed order so that the agents can handle them correctly.

For the sake of simplicity, this version of system adopts a KQML-like format for messages. The major difference between the customised format and the KQML format is that there is not striction on what attributes must be in a message. The rationale for this is that removing unnecessary message attributes such as from, to, language and ontology may reduce the overhead of message parsing and that of network communications. However, this sacrifices the generosity of the system and may make the maintenance and evolution of the system harder. Nevertheless, the second version of the system will adopt a strict KQML message format.

The performatives of messages used by this system are listed in Table 4.1. The first three performatives in the table are used for agent-server communications and the rest are used for agent collaboration. Figure 4.3 gives a typical conversation between two agents. In Section 4.2.4 I will explain the underlying architecture of each reasoning agent and how they can handle the messages in the correct order.

4.2.3 The Central Directory Server

Each system has a single directory server agent that centralises the advertisements of all the agents in the group. The directory server (server) is a persistent process running on a host known to all the reasoning agents (agents). When a new agent joins the group, it will first
<table>
<thead>
<tr>
<th>Performatives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>Used by an agent to inform the server that it wishes to join the group.</td>
</tr>
<tr>
<td>reject</td>
<td>The server informs the new agent that it is not allow to join the group at this time.</td>
</tr>
<tr>
<td>unregister</td>
<td>Used by an agent to inform the server that it wishes to leave the group.</td>
</tr>
<tr>
<td>advertise</td>
<td>Used by an agent to tell the server what goals it may be able to prove/solve.</td>
</tr>
<tr>
<td>unadvertise</td>
<td>Used by an agent to tell the server that it is not longer able to prove/solve the list of goals.</td>
</tr>
<tr>
<td>standby_stream_all</td>
<td>This combines the effects of the standby and the stream_all performatives of KQML. It is used when an agent wants a helper to find all of the answers for a given goal but return the results one by one upon request (i.e. receives a next message).</td>
</tr>
<tr>
<td>ready</td>
<td>Used by a helper agent to inform the helpee about its readiness to prove a given goal.</td>
</tr>
<tr>
<td>sorry</td>
<td>Used by a helper agent to inform the helpee that it cannot help in proving the goal.</td>
</tr>
<tr>
<td>next</td>
<td>Used by an agent to tell a helper to return the next available answer to the given goal.</td>
</tr>
<tr>
<td>tell</td>
<td>Used by a helper to send an answer to the goal back to the helpee. It is also used by the directory server to send a list of potential helpers to a reasoning agent.</td>
</tr>
<tr>
<td>eos</td>
<td>Used by a helper to inform the helpee that there is no more available answers for the given goal (e.g. End Of Stream).</td>
</tr>
<tr>
<td>discard</td>
<td>Used by an agent to inform its helper that it is no longer interested in any further answer to a previous goal. In this case, the helper can discard any previous produced answers.</td>
</tr>
</tbody>
</table>

Table 4.1: Performatives Used for System Version 1 Messaging
register with the server, and send to the server its advertisements (e.g. a list of predicates it knows and it is willing to prove/solve the goals that match those predicates). Each agent can also make or withdraw advertisements while it is in the group. After an agent leaves, the server will remove all of its advertisements. Figure 4.4, Figure 4.5, and Figure 4.6 show the interactions between the an agent and the server.

The server maintains the advertisements as tuples \( < P, As > \), where \( P \) is a predicate and \( As \) is a list of agent handles. The first time the server receives an advertisement about a predicate \( P \), it will create a new tuple for \( P \) and the \( As \) will be a list containing only the sender’s handle. When the server receives an unadvertise message about a predicate \( P \), it will first find the tuple \( (P, As) \) and then remove the sender’s handle from \( As \). If no such tuple is found, or the sender’s handle is not in \( As \), then the DA does nothing. If after handling an unadvertise message and the result \( As \) is empty, then the server will remove the whole tuple.
During a reasoning collaboration, when an agent needs help from other agents to prove/solve a sub-goal \( G \), it will query the server by sending to it a `recommend_all` message containing \( G \). The server will look up its database and return a list \( L \) of agent handles such that \((P, L)\) is a tuple in the database and \( P \) is unifiable with \( G \) (See Figure 4.7). After receiving all the list, the agent will selectively ask some or all of the agents in the list to prove/solve the goal \( G \).

![Figure 4.7: Agent Querying Server](image.png)

### 4.2.4 Reasoning Agents

The reasoning agents are the entities who perform the real reasoning tasks in the system. An agent has the following characteristics:

- It maintains its knowledge base (e.g. a set of rules and integrity constraints). For some agents such as robots, they can update the database regularly (e.g. after performing some actions and receiving perceptions).
- It has the reasoning capability. It may be “equipped” with one or more reasoning “tools” (e.g. Meta-Interpreters (See Chapter 5)).
- It can communicate with the directory server as well as other discovered agents in the group. It can also receive top level goals from external agents and send back the final answers.
- It is multi-threaded and multi-tasking. It has a Coordinator Thread (CT) for top level requests handling, a number of Worker Threads (WT) managing different reasoning tasks concurrently and independently. For each WT, there is a corresponding Reasoning Thread (RT) and a Timeout Handler Thread (HT). The WT is responsible for communicating with a helpee agent (the agent requests the task). It receives commands (e.g. `next`) from the helpee agent, fetches answers produced by the reasoning thread and sends them back to the helpee agent. The RT is the one does the actual reasoning and produces answers. The WT and RT form a Consumer-Producer relationship and the products (the answers) are placed in a buffer called `blackboard`. The HT is merely responsible for “killing” the RT if it has reasoned too long in order to free the necessary resources. In addition, each RT may request for help from a number of other agents via the Broker Threads (BT).

Figure 4.8 briefly shows what is going on inside a reasoning agent during collaboration. In the following sections I will describe the agent internal architecture in details.
Handling Incoming Requests

Figure 4.8 shows the interactions inside an agent after it receives an incoming request. The coordinator thread is responsible for the following:

1. Perform initialisation work when the agent starts. E.g. Loads the initial database, registers with the directory server, etc.
2. Receive incoming requests, preprocess them and create worker threads to handle them independently.
3. Perform finalisation work when the agent shuts down. E.g. Inform the directory server and clean up everything.

An agent may have multiple worker threads running simultaneously. Once a request is received, a new worker will be created to take over it. The worker thread starts with forking two further threads – the reasoning thread to do the reasoning job and produce answers, and the timeout handler thread. After then, the worker thread will send a ready message to the helpee agent indicating that it has accepted the request.

Usually, the goal and other parameters of the reasoning task is within the request message. The reasoning thread will first extract them and then process it according to the reasoning type (e.g. run a meta-interpreter with the given goal and parameters). Once an answer/result is produced, the reasoning thread will put it on the blackboard. Since the blackboard is shared by different reasoning threads to buffer the their answers/results, it contains tuples of the form \( <A, R> \), where \( A \) is the handle of the helpee agent and \( R \) is the answer/result. All the answers/results are recorded in sequence. When the worker thread receives a next message...
from its helpee, it will take the first result belonging to the helpee off the blackboard and send it back.

The rationale for having such a producer-consumer relation between the worker thread and the reasoning thread is to isolate the reasoning process from the helpee agent, so the reasoning thread can concentrate on its work without being interrupted by the helpee agent asking to send back the answers/results. However, this has the implication that the reasoning thread is trying to search for all of the possible answers/results to the given goal. Once it has finished the search, it will put a special tuple \(< A, eos >\) to the blackboard, which means that there is no more answer/result can be produced for \(A\). If the worker thread receives the next message from \(A\) but the answer/result taken off the blackboard is \(eos\), it will reply \(A\) with the eos message. On the other hand, if the worker thread receives a discard message from \(A\), it will erase all the answers/results for \(A\) from the blackboard, and kill the reasoning thread if it is still running.

The timeout handler thread is a very simple thread that sleeps for a certain amount of time, and then wakes up and sends a termination signal to the reasoning thread if it is still running. If the reasoning thread is terminated in this way, the special tuple \(< A, eos >\) will be put on the blackboard too.

Since the reasoning thread (RT) can may die “naturally” (i.e. finishes before timeout) or be killed by the worker thread (WT) or the timeout handler thread (HT), we need to make sure no race conditions should arise. We need to handle the following three cases differently:

- The RT exits before timeout (See Figure 4.10): In this case, the RT will kill the HT explicitly and put \(< A, eos >\) on the blackboard.
- The HT wakes up before the RT exits (e.g. timeout. See Figure 4.11): In this case, the HT will send a Terminate signal to the RT asking it to put \(< A, eos >\) on the blackboard and exit. The previous answers/results produced by the RT will remain to the blackboard.
• The WT receives a discard message from the helpee agent and the RT is still running (See Figure 4.12): In this case, the WT will send a Terminate signal to the RT asking it to kill the asleep HT and exit. The WT will also remove any answer/result for the helpee agent from the blackboard.

It is worth noting that after a reasoning thread is asked to terminate, it will also send a discard message to all the broker threads (See next section) that it is still consulting.

Figure 4.10: System Version 1 – Reasoning Thread Died Naturally

Figure 4.11: System Version 1 – Reasoning Thread Killed Due to Timeout

Getting Helps From Other Agents

As already explained, an agent may query the directory server and communicate with other agents during its reasoning process. This section will describe in details the internal interactions when an agent is asking for help. Figure 4.13 shows an example, where an agent RA discovered agent RA1 and agent RA2 via the directory server and asked them to help in solving a goal.
During the execution of a reasoning thread, it may get stuck of solving a sub-goal. In that case, the reasoning thread will query the directory server for a list of potential helper for the sub-goal (See Section 4.2.3). Often, it needs to ask only a subset of the recommended agents about the goal, e.g. it may be in the list too. After the reasoning have decided who to ask (for help), it will create a broker thread to handle the request-response communications.

When a broker thread is created, it is passed the list of agent handles and the goal with some necessary parameters. The broker thread will try to send a request to all the agents in the list. After receiving all the ready messages from the successfully connected agents\(^1\), the broker send a next message to them immediately so that the agents may return the first answer/result as soon as possible for buffering. Finally, the broker thread will reply to the reasoning thread with a ready message indicating that it has established connection with at least one helper and the results (if any) are on the way. Should none of the agent returns a ready message, the broker thread will reply to the reasoning thread with sorry which means that no helper will help with the goal.

At this point, all the helper agents are trying to solve the goal simultaneously. Since all the helpers are sent a next at the beginning, the first ever found answer/result will be sent to the broker thread straight away. The answers will be placed in the broker thread’s message buffer maintained by the ICM daemon (See Section 4.2.2). If the reasoning thread wants a result from the helpers (either for the first time or during backtracking), it will send a next message to the broker thread. The broker thread will then search the message buffer for a result. There are three cases:

- If a non-eos message is found (See Figure 4.14), the broker thread will extract the answer/result from the message and send it to the reasoning thread. The broker thread will also send another next to the helper agent who has sent the fetched result, so that if there is another answer/result produced by that agent, it will be returned and buffered.
- If an eos message is found, the broker thread will “disconnect” from the helper agent who has sent the message, and then continue the search process. However, if there is no more helper agent connected, the broker thread will return an eos message to the reasoning thread.

\(^1\)precisely, the worker threads of the agents
If there is no message in the buffer, the broker thread will suspend until an answer/result message is arrived.

The rationale of having a separate broker thread to handle the communications is to isolate the reasoning thread from the helper agents. Answers/results retrieval and buffering will be done by the broker thread. From the reasoning thread's point of view, a broker thread is the one who can provide answers/results for a "stuck sub-goal" when they are needed, and different broker threads may be active and buffering results for different sub-goals at the same time. While the reasoning thread backtracks to a "stuck sub-goal", it will simply ask the corresponding broker thread to return the next available answer/result. If no more answer/result is available (i.e. receives the eos message from the broker thread),
the reasoning thread will fail the current “stuck sub-goal” and continue the backtrack. Any broker thread will exit automatically after sending out an eos message to the reasoning thread.

In addition, a broker thread will try to buffer one but only one answer/result from each helper agent (e.g. send the next next message to a helper only after its previously returned answer/result is used). This design prevents the helper agents from sending back all of the answers/results as soon as they are found, while still allowing the broker thread to buffer at most $N$ (the number of still connected helpers) answers/results locally. In another word, the design can reduce network traffic as well as maintaining the reasoning performance (e.g. the reasoning thread can continue the reasoning when the broker thread is fetching the next answer/result).

Sometimes the reasoning thread may decide not to need any further answers/results from the helpers (e.g. it is killed by the timeout handler thread or the worker thread), it will send a discard message to the broker thread. Upon receiving such a message, the broker thread will forward the discard message to all the still connected helper agents and clean the message buffer.

4.3 System Version 2

This version is the second attempt design to the system, and is largely based on the first version. Here are the major differences between the two versions of the system:

Agent Communication In this version, I used the publish/subscribe mechanism for inter-agent communications. Also, strict KQML message format is adopted to make the system more generic.

Agent Discovery This is no central directory server anymore. Each agent will maintain a local directory. Advertisements are broadcasted to all the agents in the group.

The second version of the system has been implemented with QuProlog 8.0. In the following sections, I will explain the changes in details.

4.3.1 Architecture Overview

All the agents in the system have the same internal architecture but different knowledge bases. These agents form a loosely coupled group. When a new agent joins the group, it will first broadcast a register message to the existing group, and then broadcast its advertisements. When an existing agent receives a register message, it will send all of its advertisements to the new agent directly. (See Figures 4.15).

If an existing agent updates its reasoning capability, it can also broadcast the new advertise or unadvertise message to the whole group. When an existing agent leaves the group, it will broadcast an unregister message.

With this new architecture, each agent in the system will be able to discover potential helpers about a sub-goal by query its local directory. The agent can then send the sub-goal to the helpers via peer-to-peer messaging.

4.3.2 Inter-agent Communication

In this version of the system, there are changes in both the high level and low level communications. For high level communication, it uses the standardised KQML format in order
CHAPTER 4. DISTRIBUTED REASONING SYSTEM

(a) New Agent Broadcasts register

(b) Existing Agents Reply with advertise

(c) New Agent Broadcasts advertise

Figure 4.15: System Version 2 – New Agent Arrives

to provide a simple and uniform API for developing meta-interpreters for different applications. For low level communication, since the central directory server is removed, the agents will need a way to broadcast information among the whole group. This is done via the public-subscribe mechanism.

Low Level Communication

There are two basic types of messaging – Peer-to-Peer and Broadcast. They are done via the Pedro server (See Section 2.8.3). Like ICM, the Pedro server maintains a message buffer for each registered agent.

Each agent can “publish” a message to the Pedro server. If an agent is interested in receiving a certain type of messages, it will need to “subscribe” it from the Pedro server first. After the subscription is made, any further published messages matching that type will be placed into the agent’s message buffer by the server, and the agent can fetch them anytime it wants. For example, in the system, each agent subscribes to the advertise messages. When

\footnote{registered with the server, not registered with the agent group. When a QuProlog process starts, it will register to the given Pedro server automatically.}
an agent publishes its advertisement to the Pedro server, the advertisement will be delivered to all the agents in the group. This is broadcasting.

In the current implementation, peer-to-peer messaging is also done via the Pedro server. By default, each agent subscribes all the messages with itself as the the destination. If agent 1 wants to send a message to agent 2, it will send the message to the Pedro server, and the Pedro server will “route” it to agent 1’s message buffer (See Figure 4.16).

![Figure 4.16: Agent Communication via Pedro](image)

**4.3.3 Reasoning Agents**

The internal architecture has two minor changes:

- It maintains a local directory. The content of the directory is the same as that of the central directory server described in the system version 1.

- It has a *directory thread* (DT) for maintaining the local directory.

The directory thread is a persistent thread created by the coordinator thread:

1. On agent startup, it will subscribe four types of messages from the Pedro server – *register*, *unregister*, *advertise* and *unadvertise*. Then it will broadcast *register* and its advertisement.

2. When it receives a *register* message from another agent, it will send its advertisement to that agent via a peer-to-peer message.

3. When it receives an *advertise* or *unadvertise* message from another agent, it will update the local directory accordingly, just like the central directory server in the old system does.

4. When it receives an *unregister* message from another agent, it will remove all the advertisements of that agent from the local directory.

5. On agent shutdown, it will broadcast *unregister*.

Since each agent maintains a local directory, when a reasoning thread gets stuck on a sub-goal, it only needs to query the local directory instead of the central directory server in order to find out a list of potential helpers. This can reduce the network traffic greatly.
4.4 Comparison of Two Versions

Though there are several architectural differences between the two versions of the system, their functionalities are the same, and they both have the Property 4.1, Property 4.2, and Property 4.4. The changes in version 2 have pros and cons and they are depending on the actual applications.

Central Directory vs. Local Directory

In the central directory approach, the maintenance work is relatively easy because there is only a single copy of the directory in the whole system. In the local directory approach, since every agent has its own copy of the directory, we need to make sure all these directories are consistent all the time. Also, if the directory is big, we also need to make sure each agent has the sufficient resource (e.g. memory) to hold the directory.

For performance: in the central directory approach, every time when an agent needs to ask for help, it has to query the server via a network connection. In the local directory approach, the agent only needs to query the local directory. Hence, if in an application requires lots of agent collaborations, the local directory approach can reduce the network traffic significantly.

Self-Invented Message Format vs. KQML Message Format

Using the well-known and fully documented KQML format can give generosity to the system. It will make the evolution of the system easier in the long run. It can also provide a uniform API for customising the system for different applications. However, the generic KQML format tends to make a message unnecessarily big, and the complex message structure will cause the agent to take more time to parse a message. Below are two sample messages carrying the same information:

\[
\text{stream\_all(abdemo,}
\]
\[
\quad\text{[[car\_doesn\_start(my\_car)], [lights\_go\_on(my\_car)], 4]).}
\]

and

\[
\text{stream\_all([}
\text{from(agent1),}
\text{content([}
\quad\text{task(abdemo),}
\quad\text{goal([car\_doesn\_start(my\_car)]),}
\quad\text{delta([lights\_go\_on(my\_car)]),}
\quad\text{depth(4)}
\quad\text{])}
\text{]).}
\]

The first message is used for system version 1 and its structure is assumed to be understood by both the sender and receiver – the first argument of \text{stream\_all}/2 is the reasoning type and the second argument is a list of parameters such as the goal, the current delta and the search depth. The second message has the KQML format and is used for the system version 2. It is easy to see that the second message is bigger and more complex, but more generic.
ICM vs. Pedro

The two approaches are described in Section 4.2.2 and Section 4.3.2. One observation for peer-to-peer messaging in system version 2 is that a message has to be sent from the source agent to the Pedro server, and then sent from the server to the destination agent. This seems to increase the network traffic and the message delay. In order to verify that, I did a benchmarking between ICM and Pedro. Table 4.2 shows the result.

<table>
<thead>
<tr>
<th>Message Payload</th>
<th>Average Time Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 1 kilobyte, 3000 rounds</td>
<td>2 seconds</td>
</tr>
<tr>
<td>&gt; 1 kilobyte, 3000 rounds</td>
<td>41 seconds</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmarking Between ICM and Pedro: Two peers and the Pedro server were running on different machines on different sub-nets

The benchmarking result is quite surprising, it didn’t show that peer-to-peer messaging with Pedro is slower than that with ICM. I guess this is because the implementations of Pedro is more efficient than that of ICM.

4.4.1 Conclusion

After comparing the differences between the two versions of the system, I conclude that:

1. The central directory approach is good for applications where agent device’s memory is limited, or agent collaboration is occasional. The local directory approach is good for applications where bandwidth is low and agent collaboration is frequent.

2. Adopt the KQML message format for system prototyping, and use customised message formats to improve system performance.

3. Use Pedro instead of ICM.

4.5 Example of Multi-Agent Reasoning

In this section I will show you a very simple example of having a group of agents to perform deductive reasoning collaboratively.

4.5.1 A Distributed Deductive Meta-Interpreter

First of all, let’s give each of the agent a tool – a distributed deductive meta-interpreter (Listing 4.1). The distributed meta-interpreter is a modified version of the demo meta-interpreter in Section 2.2.2.

```
1 %
2 %% =========== A Distributed Deductive ===========
3 %% ========= Reasoning Meta-Interpreter =========
4 %
5 do_reasoning(g_demo, List, Client) :-
```
agent_handle(Hdl),
member(goal(Goal), List),
member(dont_ask(DontAsk), List), !,
forall(
    g_demo(Goal, Expl, [Hdl|DontAsk]),
    (assertz(result_for(Client, [goal(Goal), result(Expl)])),
    ),
    assertz(result_for(Client, [])).

g_demo([], [], _).
g_demo([G|Gs], [E|Es], DontAsk) :-
g_demo_one(G, E, DontAsk),
agent_handle(Hdl),
g_demo(Gs, Es, [Hdl]).

g_demo_one(G, (G, Es), _) :-
rule(G, Gs),
(Gs == [] ->
Es = true;
(agent_handle(Hdl),
g_demo(Gs, Es, [Hdl])
))
).

g_demo_one(G, (G, Es), DontAsk) :-
find_helpers(G, Agents),
diff_list(Agents, DontAsk, Helpers), !,
Helpers \== [],
union_list(DontAsk, Helpers, NewDontAsk),
TaskList = [content([task(g_demo),
goal([G]),
dont_ask(NewDontAsk)])],
rt_recruit_broker(Helpers, TaskList, Broker), !,
repeat,
next([]) ->> Broker,
Result <=< Broker,
(eos(_)) = Result ->
( retract(consulting(_, Broker)),
!, fail);
(tell(List) = Result,
member(content(Content), List),
member(goal([G]), Content),

I will explain the meta-interpreter briefly:

**The top level stub**
The `do_reasoning/3` predicate defined from line 6 ~ 17 is a top level stub called by a reasoning thread. It has the following behavior:

1. It extracts the goal and the `DontAsk` parameters from a KQML request message.
2. It calls the `g_demo` to perform deductive reasoning *globally* (e.g. may ask others if stuck) on the given goal with the extra parameter `DontAsk`.
3. For each answer it finds, it will put it to the blackboard (line 14) so that the worker thread can fetch it later if asked to.
4. After the reasoning thread finishes the search, it puts `empty` to the blackboard (line 17) saying that there will be no more answer available.

Here, the `DontAsk` parameter is a list of agent handles given by the requester. It means that “If you cannot prove this goal or break this goal into a list of sub goals, then don’t bother to ask the people in the list for help”. This parameter is used for preventing cyclically asking for help between the agents.

**Global deductive reasoning**
This is done by calling `g_demo`. The `g_demo` in line 19, 20 and `g_demo_one` are similar to `demo` and `demo_one`. The only behavioral difference is that `g_demo` and `g_demo_one` will change the extra parameter `DontAsk` to be a list containing only itself. This is to say “for the sub-goals of mine, I am free to ask whoever I want (remotely) except myself”.

**Asking for help**
The `g_demo_one` in line 35 is the one we are interested the most. It is called when the current agent fails to prove a goal `G` alone. It does the following:

- In line 36, it either queries the central directory server or searches its local directory (depending on which version of the system we are using) for a list of potential helpers.
- Line 37 is to find out a list of agents that are allowed to ask, and line 38 makes sure there is at least someone to ask (otherwise, don’t bother to send out any request).
- Line 39 is to generate the new `DontAsk` for the helpers and line 40 ~ 44 is to create the request message.
- In line 46, the current agent recruits a broker to forward the request to the helpers. The broker will buffer any returned answers.
• The rest is to try to get the next answer from broker. If an answer is returned from the broker, the current agent will extract it from the message, and use it to continue the reasoning process. Upon backtrack, the current agent will try to ask another answer from the broker. If no more answer is available, then demo one fails.

4.5.2 The Sample Database – Family Tree

Now we can try to test the system! Let’s assume that we have only two agents in the group, and their knowledge is listed in Appendix A.2.

On start up, each agent will publish an advertisement containing all it knows. For example, agent 2 will advertise that it can help to solve any goal matching parent/2, male/1, grandparent/2, father/2 and predecessor/2.

If we send a request ?- grandparent(pam, X) to agent 1, we will get back the results grandparent(pam, ann) and grandparent(pam, pat). The log of the collaboration is:

A1: ?- grandparent(pam, X).
A1: ?- parent(pam, Y), parent(Y, X). % resolve it using the rule
A1: ?- parent(bob, X). % it found parent(pam, bob).
A1 asks A2: ?- parent(bob, X).
A2 replies A1: parent(bob, ann).
A1: ?- [] % succeeded with grandparent(pam, ann).
A1 asks A2: next
A2 replies A1: parent(bob, pat).
A1: ?- [] % succeeded with grandparent(pam, pat).
A1 asks A2: next
A2 replies A1: eos % A2 said no more answers. Notice that
% even though A1 had advertised parent/2,
% A2 couldn’t ask A1 because A1 was in
% the Don’tAsk list.
A1: no. % A1 backtracked and failed eventually.
Chapter 5

Distributed Abductive Meta-Interpreters

Chapter 4 presents a multi-agent system for coordinating a collection of agents to perform reasoning tasks. In order to give the agents the ability to perform collaborative abductive reasoning from their knowledge, I have also proposed two abductive meta-interpreters. The first one is a general purpose meta-interpreter based on the Kakas-Mancarella proof procedure. The second one is a simple abductive planner for planning with simplified event calculus.

Since the multi-agent system and the meta-interpreter relies on each other to achieve the objective of letting agents to perform distributed abduction, we call them Distributed Abductive REasoning System (DARES).

This chapter describes the two meta-interpreters in two sections.

5.1 General Purpose Distributed Abductive Meta-Interpreter

5.1.1 Scenario

In some applications, agents may need to collaborate to explain a given goal $G$. Since each agent will have its knowledge base $T_n$ and set of integrity constraints $I_n$, let $\Delta$ be the final explanation (result) obtained, and $As$ be the cluster of agents that collaborated together to give $\Delta$ (each agent must either have resolved a non-abducible sub-goal or have abducted a new abducible sub-goal). Assuming that:

- All the agents in the group agree to the set of abducible predicates.
- $\bigcup_{n \in As} T_n$ is consistent.

$\Delta$ must satisfy the following:

- $\bigcup_{n \in As} T_n \cup \Delta \models G$;
- $\bigcup_{n \in As} T_n \cup \Delta$ is consistent;
- $\bigcup_{n \in As} T_n \cup \Delta$ satisfies $\bigcup_{n \in As} I_n$
This implies that the result obtained from the cluster of agents must not violate the integrity constraints of each agent in the cluster, i.e. satisfies Property 4.3 (See Section 4.1.4).

Let’s look at a very simple diagnosis example with two agents. The set of abducible predicates is \{battery\_flat(X), has\_no\_fuel(X), broken\_indicator(X)\}. The knowledge base and integrity constraints of each agent is listed below:

**Agent 1**

\[
\text{car\_doesnt\_start}(X) \leftarrow \text{has\_no\_fuel}(X).
\]

\[
\text{lights\_go\_on}(\text{mycar}).
\]

\[
\text{has\_no\_fuel}(X), \text{not fuel\_indicator\_empty}(X), \text{not broken\_indicator}(X).
\]

**Agent 2**

\[
\text{car\_doesnt\_start}(X) \leftarrow \text{battery\_flat}(X).
\]

\[
\text{fuel\_indicator\_empty}(\text{mycar}).
\]

\[
\text{battery\_flat}(X), \text{lights\_go\_on}(X).
\]

If we ask the agents to explain \text{car\_doesnt\_start}(\text{mycar}), any of the follow answers would satisfy the requirements:

1. \(A_s = [\text{agent1}], \Delta = \{\text{broken\_indicator}(\text{mycar}), \text{has\_no\_fuel}(\text{mycar})\}\)
2. \(A_s = [\text{agent2}], \Delta = \{\text{flat\_battery}(\text{mycar})\}\)
3. \(A_s = [\text{agent1}, \text{agent2}], \Delta = \{\text{broken\_indicator}(\text{mycar}), \text{has\_no\_fuel}(\text{mycar})\}\)

### 5.1.2 Proof Procedure

In this section explains the proof procedure for the meta-interpreter.

Same as before, abducibles are divided into two sets: base abducibles and non-base abducibles. A base abducible is a positive literal or a negative literal with abducible predicate. A non-base abducible is a negative literal with a non-abducible predicate. Non-abducibles are positive literals with non-abducible predicates.

\(G\) is a list of ground sub-goals to be explained and has the form \(\leftarrow L_1, \ldots, L_k\), where \(L_i\) is either a positive or negative literal. If \(k = 0\) then \(G = []\). The complement of \(L\) is written as \(L^*\). A top-down proof procedure interleaves the global abductive derivation, the global consistency derivation, the local abductive derivation and the local consistency derivation, to reduce \(G\) to \([],\) (the empty list). The \(\Delta\) (a set of ground abducibles) is collected and \(A_s\) (the set of the agents in the cluster) is expanded along the procedure. Initially, \(\Delta\) is empty and \(A_s\) contains only the current agent.

**Global Abductive Derivation**

The global abductive derivation succeeds if \(G\) is empty. Otherwise, \(G'\) is obtained by removing a literal \(L\) from \(G\), \(\Delta'\) and \(A_s'\) are the set of ground abducibles and the agent cluster obtained after applying one of the following rules:
1. If \( L \) is a non-abducible. If a rule whose head can match \( L \) exists, and the instantiated body is \( C \), then the current agent continues the global abductive derivation on \( C \cup G' \) with \( \Delta' = \Delta \) and \( As' = As \).

2. If \( L \) is a non-abducible and the previous rule fails, the current agent will try to ask a helper agent \( A \) to perform a global abductive derivation on ← \( L \). If \( A \in As \) then \( A \) will perform the global abductive derivation on ← \( L \) with \( \Delta' = \Delta \) and \( As' = As \). If \( A \not\in As \), then \( A \) will first perform a global consistency derivation on \( \Delta \) with \( A \cup As \), and then perform a global abductive derivation on ← \( L \) with \( \Delta'' \) and \( A \cup As \), where \( \Delta'' \) is obtained after a successful global consistency derivation.

   - If \( A \) succeeds in a global abductive derivation and returns \( \Delta' \) and \( As' \), then the current agent continues the current global abductive derivation on \( G' \) with \( \Delta' \) and \( As' \).
   - otherwise, the current derivation fails.

3. If \( L \) is an abducible and \( L^* \) is already in \( \Delta \), the derivation fails.

4. If \( L \) is an abducible and \( L \) is already in \( \Delta \), then the current agent continues the global abductive derivation on \( G' \) with \( \Delta' = \Delta \) and \( As' = As \).

5. If \( L \) is a base abducible and neither \( L \) nor \( L^* \) is in \( \Delta \), if there exists a global consistency derivation on \( \{ L \} \cup \Delta \) with \( As \), and \( \Delta' \) is obtained after the global consistency derivation, then the current agent continues the global abductive derivation on \( G' \) with \( \Delta' \) and \( As' = As \).

6. If \( L \) is a non-base abducible and \( L \) is not in \( \Delta \), if there exists a local consistency derivation on \( \{ \leftarrow L^* \} \) with \( \Delta'' = \Delta \cup \{ L \} \), and if there exists a global consistency derivation on \( \Delta'' \) with \( As \), where \( \Delta'' \) is obtained after the local consistency derivation, then the current agent continues the global abductive derivation on \( G' \) with \( \Delta' \) and \( As' = As \) where \( \Delta' \) is obtained after the global consistency derivation. Otherwise, the current derivation fails.

Global Consistency Derivation

In global consistency derivation, \( \Delta_0 \) is the given set of abducibles and \( As \) is the set of agents in the cluster.

\( \Delta' \) is obtained by passing \( \Delta \) around the agents in \( As \) exactly once. The agents in \( As \) are labeled as \( A_1, \ldots, A_n \). For one round consistency check on \( \Delta \):

1. At first, \( \Delta \) is passed to \( A_1 \). If there is a local abductive derivation on \( \Delta \) (\( \Delta \) is considered as a list of goals) by \( A_1 \) and \( \Delta_1 \) is obtained after the derivation, then \( \Delta_1 \) is passed to \( A_2 \).

2. Finally, if \( \Delta_{n-1} \) is passed to \( A_n \) and \( \Delta' \) is obtained after a successful local abductive derivation by \( A_n \) on \( \Delta_{n-1} \), then the current round of consistency check finishes.

If one round consistency check finishes and \( \Delta' = \Delta \), then the global consistency derivation succeeds. If one round consistency check finishes but \( \Delta' \neq \Delta \) (i.e. \( \Delta \) expands to become \( \Delta' \)), then start another round of consistency check. The global consistency check continues until \( \Delta \) remains the unexpanded after one successful round of consistency check. The final \( \Delta' \) obtained is the result for the successful global consistency derivation.

If one round consistency check cannot finishes (i.e. \( \Delta \) cannot satisfy an agent’s integrity constraints), then the global consistency derivation fails.
Local Abductive Derivation

The local abductive derivation is similar to the abductive derivation of the Kakas-Mancarella proof procedure described in Section 2.3.3. The difference is that this one checks integrity constraints.

The local abductive derivation succeeds if $G$ is empty. Otherwise, $G'$ is obtained by removing a literal $L$ from $G$, $\Delta'$ is the set of ground abducibles obtained after applying one of the following rules:

1. If $L$ is a non-abducible. If a rule whose head can match $L$ exists, and the instantiated body is $C$, then the current agent continues the local abductive derivation on $C \cup G'$ with $\Delta' = \Delta$. Otherwise, the derivation fails.

2. If $L$ is an abducible and $L^*$ is already in $\Delta$, the derivation fails.

3. If $L$ is an abducible and $L$ is already in $\Delta$, then agent continues the local abductive derivation on $G'$ with $\Delta' = \Delta$.

4. If $L$ is a base abducible and neither $L$ nor $L^*$ is in $\Delta$, let $C$ be the set of integrity constraints containing $L$, and $C''$ be the set obtained by removing $L$ from each constraint in $C$. If there exists a successful local consistency derivation on $C''$, then the agent continues the local abductive derivation on $G'$ with $\Delta' = \{L\}$.

5. If $L$ is a non-base abducible and $L$ is not in $\Delta$, let $C$ be the set of integrity constraints containing $L$, and $C'$ be the set obtained by removing $L$ from each constraint in $C$. If there exists a local consistency derivation on $\{C' \cup \leftarrow L^*\}$ with $\Delta'' = \Delta \cup \{L\}$, then the agent continues the local abductive derivation on $G'$ with $\Delta'$ where $\Delta'$ is obtained after the local consistency derivation. Otherwise, the derivation fails.

Local Consistency Derivation

The local consistency derivation is the same as the consistency derivation of the Kakas-Mancarella proof procedure described in Section 2.3.3. I included it here for easy reading.

In the consistency derivation, $F$ is a set of goals to be checked for consistency. The consistency derivation succeeds if $F$ is empty. The consistency derivation fails if $F$ contains $\emptyset$. Otherwise, let $F' \cup G = F$ and $G'$ is obtained by removing a literal $L$ from $G$ and $\Delta'$ is the set of abducibles after applying a rule. The rules in the consistency derivation are:

1. If $L$ is a non-abducible. $C$ is the set of all the instantiated non empty bodies of the rules in the database whose heads can match $L$, continue the consistency derivation on $C \cup F'$ with $\Delta$.

2. If $L$ is abducible and $L$ is already in $\Delta$, continue the consistency derivation on $F' \cup G'$ with $\Delta' = \Delta$.

3. If $L$ is abducible and $L^*$ is already in $\Delta$, then continue the consistency derivation with $F'$ and $\Delta' = \Delta$.

4. If $L$ is base abducible and neither $L$ nor $L^*$ is in $\Delta$, then continue the consistency derivation with $F'$ and $\Delta' = \Delta \cup L^*$.

5. If $L$ is non-base abducible and $L$ is not in $\Delta$, if there exists an successful abductive derivation on $\{\leftarrow L^*\}$ with $\Delta$, then continue the consistency derivation on $F'$ with $\Delta'$ where $\Delta'$ is obtained from the abductive derivation.
Example Run of the Meta-Interpreter

Let's try out the example described in Section 5.1.1 with the distributed reasoning system and the meta-interpreter.

When we asked agent 1 to explain `car_doesnt_start(mycar)`, we got $As = \{\text{agent1}\}$, $\Delta = \{\text{broken_indicator(mycar)}, \text{has_no_fuel(mycar)}\}$:

1. Initially, only agent 1 was in the cluster.
2. Agent 1 resolved the non-abducible `car_doesnt_start(mycar)` to `has_no_fuel(mycar)`, and then tried to check the integrity constraint $\leftarrow \text{has_no_fuel(mycar)}, \text{not fuel_indicator_empty(mycar)}, \text{not broken_indicator(mycar)}$.
3. Agent 1 could not fail to prove the non-base abducible `not fuel_indicator_empty(mycar)`, so it abduced `broken_indicator(mycar)`.
4. Since agent 1 was still the only one in the cluster, and the result $\Delta = \{\text{broken_indicator(mycar)}, \text{has_no_fuel(mycar)}\}$ is consistent with its knowledge base and integrity constraint, it was returned.
5. Then agent 1 backtracked and asked agent 2 to prove `car_doesnt_start(mycar)`.
6. Agent 2 abduced `battery_flat(mycar)` in the global abductive derivation and abduced `not lights_go_on` in the local consistency derivation. However, during the global consistency derivation, agent 1 failed to prove `not lights_go_on(mycar)` so it rejected it.
7. Agent 2 backtracked and couldn’t find any more answer. So none answer was returned from agent 2 to agent 1.
8. Finally, agent 1 finished the backtracking without finding any more results.

When we asked agent 2 to explain `car_doesnt_start(mycar)`, we received $As = \{\text{agent2}\}$, $\Delta = \{\text{battery_flat(mycar)}\}$

, and

$As = \{\text{agent1, agent2}\}, \Delta = \{\text{broken_indicator(mycar)}, \text{has_no_fuel(mycar)}\}$

:

1. Initially, only agent 2 was in the cluster.
2. Agent 2 resolved `car_doesnt_start(mycar)` to `battery_flat(mycar)`, and abduced `battery_flat(mycar)` in its local consistency check, it didn’t know `lights_go_on(mycar)` so the integrity constraint $\leftarrow \text{battery_flat(mycar)}, \text{lights_go_on(mycar)}$ is consistent. Agent 2 returned $As = \{\text{agent2}\}, \Delta = \{\text{battery_flat(mycar)}\}$ as the first answer.
3. Agent 2 backtracked and asked agent 1 to prove `car_doesnt_start(mycar)`. Agent abduced `broken_indicator(mycar)` and `has_no_fuel(mycar)`, and they passed the global consistency check between agent 1 and agent 2, so $As = \{\text{agent1, agent2}\}, \Delta = \{\text{broken_indicator(mycar)}, \text{has_no_fuel(mycar)}\}$ was returned from agent 1 to agent 2, and agent 2 returned it to us as the second answer.
4. Finally, agent 2 finished the backracking without finding any more results.

Clearly, those answers satisfied the requirements. Furthermore, it can be observed that the more agents in the cluster, the better "quality" the returned answer is. Hence, for a real application, we may do the following:

- Ask the agents to returned a number of answers, and select the one with the cluster contain the most agents.
- Ask the an agent to prove a goal with a predefined cluster, i.e. force the agent to check all of its answers with the given "experts" before returning them.

5.1.3 Correctness of the Procedure

In this section, I attempt to show that the proof procedure I have proposed indeed satisfies Property 4.3.

First of all, the local abductive derivation and the local consistent derivation are basically the Kakas-Mancarella proof procedure with integrity constraint check. So their correctness is based on Kakas's work.[17]

Lemma 5.1 The local abductive derivation and the local consistent derivation together will give a correct answer that satisfies the agent’s local integrity constraints.

Secondly, the Δ obtained after a global consistent derivation satisfies all the integrity constraints of all the agents in the cluster:

1. When an agent is passed a Δ, it will consider it as a goal and perform a local abduction on it. Since the Δ contains only abducibles, the local abduction will have the same effect of checking whether the abducibles are consistent with the agent’s database and satisfy the agent’s integrity constraints.

2. If the Δ expands in a round of global consistency, it will be checked for another round. Therefore, all the agents will have the chance to check every new abducibles abduced by an agent after them.

3. If one round is finished and the Δ doesn’t change, then the Δ must be consistent with every agent’s database and satisfies every agent’s integrity constraint (by Lemma 5.1).

4. The global consistency check will eventually terminate because the set of all ground abducibles is finite.

Lemma 5.2 The Δ obtained by a successful global consistent derivation will satisfy all the integrity constraints of all the agents in the cluster. I.e. it satisfies Property 4.3.

Finally, I need to show that the result produced by a successful global abductive derivation will satisfy all the integrity constraints of all the agents in the cluster. This is done as follow:

1. Base case: Let A be an agent who has never asked any other agent to resolve a sub-goal during the global abductive derivation, i.e. rule 2 is never selected. For A, the Δ would change only after applying rule 5 or rule 6. But the Δ’ obtained after applying rule 5 or rule 6 must have gone through a global consistency check. So (by Lemma 5.2) the Δ return from A after a successful global abductive derivation satisfy Property 4.3.
2. **Induction Assumption**: The current $\Delta$ is consistent with the integrity constraints of each agent in the current cluster.

3. **Induction Step**: Let $A'$ be an agent who has asked some other agents to help in resolving some sub-goals, i.e. rule 2 is applied (one or more times), and all the $\Delta'$ returned from the helper agents satisfies Property 4.3 (If the agent being asked is a new agent to the cluster, according to the global abductive derivation, it will first perform a global consistency derivation before starting the global abductive derivation. The global consistency derivation guarantees the new agent joins the cluster consistently). Similarly to the base case, the $\Delta$ would ever change again after applying rule 5 or 6. But the global consistency derivation in these rules will guarantee that changed $\Delta$ satisfy Property 4.3. Therefore, the result returned from $A'$ will satisfy Property 4.3 too.

In conclusion:

**Lemma 5.3** The $\Delta$ obtained by a successful global abductive derivation will satisfy all the integrity constraints of all the agents in the cluster. I.e. it satisfies Property 4.3.

and therefore, my proposed proof procedure (and the meta-interpreter) will also satisfies Property 4.3.

### 5.1.4 Optimisation

When implementing the meta-interpreter, there are several optimisations possible. One major optimisation I made to the meta-interpreter is in the global consistency derivation.

In the original proof procedure, we can observe that during the each round of consistency check among the agent in the cluster, each agent needs to perform a local abduction using $\Delta$ as the goal. If one new abducible is added to the $\Delta$, then in the next round, each agents will need to perform another local abduction on the expanded $\Delta$. In fact, in each round, we only expect each agent to check the newly added abducibles for consistency, i.e. perform local abduction using the set of newly added abducibles as the goal and the original $\Delta$ as the input $\Delta$. Therefore, in my optimised version of the meta-interpreter:

1. In the four derivations, current $\Delta$ is divided into two disjunctive subset – $\Delta_{FIXED}$ and $\Delta_{NEW}$. $\Delta_{FIXED}$ is the set of abducibles given to each derivation at the beginning and the set will remain unchanged during the current derivation. Any abducibles added during this derivation will be put into $\Delta_{NEW}$. When the derivation succeeds, $\Delta' = \Delta_{FIXED} \cup \Delta_{NEW}$.

2. Whenever a new abducible $L$ is about to add to $\Delta_{NEW}$ in the global abductive derivation, a global consistency derivation will start with $\Delta'_{FIXED} = \Delta_{FIXED} \cup \Delta_{NEW}$ and $\{L\}$:
   
   (a) In the first round, agent $A_1$ will perform a local abduction with $\{L\}$ as the goal, $\Delta'_{FIXED}$ as the input fixed set of abducibles. If the local abduction succeeds, and $\Delta'_{NEW_1}$ is the output set of (new) abducibles, then $\Delta'_{FIXED}$ and $\Delta'_{NEW_1}$ will be passed to the next agent.

   (b) When the round is finished, if obtained $\Delta'_{NEW_n}$ is equal to $\{L\}$, then the global consistency derivation succeeds. Otherwise, we will start another round with $\Delta'_{NEW_n} - \{L\}$ as the initial goal and $\Delta'_{FIXED} \cup \{L\}$ as the new fixed set of abducibles.

Hence, the optimised meta-interpreter will do consistency check on $\Delta$ incrementally instead of brute force checking.
5.2 Distributed Abductive Event Calculus Planner

5.2.1 Scenario

Another major application area of abductive reasoning is event calculus planning (See Section 2.4). Lots of research has gone into single robot planning. There are also same examples about planning with multiple robots. But the planning is usually done by a single abductive reasoning process and the action commands of a plan are sent to different robots for execution.

In order to perform a single “central” planning, all of the robots’ knowledge (e.g. the robots’ beliefs about the world) must be put together before the planning process starts. In addition, a planning task is usually computationally hungry. This is even worst for multiple robots planning. Therefore, for the applications where the robots are loosely coupled and their actions are highly independents, we may try to break the planning task into a series of sub-tasks and distribute them to the robots with suitable capabilities. When the sub-plans for sub-tasks are returned, we can then merge them to form a top level plan. In fact, if we know that the sub-plans are consistent (i.e. there is no conflict actions between the plans), we don’t even need to robots to return them but just execute them straight away.

![Blocks World Configuration 1](image)

**Figure 5.1: Blocks World Configuration 1**

Let’s look at a simple blocks world example. As shown in Figure 5.1 four robots are working at four disjunctive regions. The regions are separated by the fences and the robots cannot move across them. However, a robot can pass an object it is holding to its neighbor. Each robot only knows about the configuration of the region it is working at. The predicates
used for modeling the robot actions and the regional states are:

<table>
<thead>
<tr>
<th>Regional States</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>top(X)</td>
<td>Object X is at the top of a stack.</td>
</tr>
<tr>
<td>on(X, Y)</td>
<td>Object X is on object Y.</td>
</tr>
<tr>
<td>neighbor(robot(X), robot(Y))</td>
<td>Robot X and robot Y are neighbors.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot States</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>holding(robot(R), X)</td>
<td>Robot R is holding an object X.</td>
</tr>
<tr>
<td>empty_hands(robot(R))</td>
<td>Robot R is not holding anything.</td>
</tr>
<tr>
<td>at_fence(F)</td>
<td>Robot R is at the fence F.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot Actions</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>pick_up(robot(R), X)</td>
<td>Robot R picks up an object X.</td>
</tr>
<tr>
<td>put_down(robot(R), X)</td>
<td>Robot R puts down the object X it is holding.</td>
</tr>
<tr>
<td>stack_on(robot(R), X, Y)</td>
<td>Robot R place the object X it is holding on top of an object Y</td>
</tr>
<tr>
<td>give(robot(R1), robot(R2), X)</td>
<td>Robot R1 gives the object X it is holding to robot R2.</td>
</tr>
</tbody>
</table>

According to Figure 5.1, the initial beliefs of each robot about its region is:

<table>
<thead>
<tr>
<th>Robot 1</th>
<th>Robot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially(top('A')).</td>
<td>initially(top('F')).</td>
</tr>
<tr>
<td>initially(on('A', 'B')).</td>
<td>neighbor(robot(2), robot(1)).</td>
</tr>
<tr>
<td>neighbor(robot(1), robot(2)).</td>
<td>neighbor(robot(2), robot(4)).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Robot 3</th>
<th>Robot 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>initially(top('E')).</td>
<td>initially(top('C')).</td>
</tr>
<tr>
<td>neighbor(robot(3), robot(1)).</td>
<td>initially(on('C', 'D')).</td>
</tr>
<tr>
<td>neighbor(robot(3), robot(4)).</td>
<td>neighbor(robot(4), robot(2)).</td>
</tr>
<tr>
<td>neighbor(robot(4), robot(3)).</td>
<td>neighbor(robot(4), robot(3)).</td>
</tr>
</tbody>
</table>

The robots can perform the same set of actions. Below is the action effects of robot 1:
CHAPTER 5. DISTRIBUTED ABDUCTIVE META-INTERPRETERS

\[
\text{initiates}(\text{pick\_up}(\text{robot}(1), X), \text{holding}(\text{robot}(1), X), T) \leftarrow \\
\quad \text{holds\_at}(\text{empty\_hands}(\text{robot}(1)), T), \\
\quad \text{holds\_at}(\text{top}(X), T).
\]

\[
\text{initiates}(\text{pick\_up}(\text{robot}(1), X), \text{top}(Y), T) \leftarrow \\
\quad \text{holds\_at}(\text{empty\_hands}(\text{robot}(1)), T), \\
\quad \text{holds\_at}(\text{on}(X, Y), T), \\
\quad \text{holds\_at}(\text{top}(X), T).
\]

\[
\text{terminates}(\text{pick\_up}(\text{robot}(1), X), \text{top}(X), T).
\]

\[
\text{terminates}(\text{pick\_up}(\text{robot}(1), X), \text{on}(X, Y), T).
\]

\[
\text{terminates}(\text{pick\_up}(\text{robot}(1), X), \text{empty\_hands}(\text{robot}(1)), T).
\]

\[
\text{initiates}(\text{put\_down}(\text{robot}(1), X), \text{top}(X), T) \leftarrow \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(1), X), T).
\]

\[
\text{initiates}(\text{put\_down}(\text{robot}(1), X), \text{empty\_hands}(\text{robot}(1)), T) \leftarrow \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(1), X), T).
\]

\[
\text{terminates}(\text{put\_down}(\text{robot}(1), X), \text{holding}(\text{robot}(1), X), T).
\]

\[
\text{terminates}(\text{put\_down}(\text{robot}(1), X), \text{empty\_hands}(\text{robot}(1)), T).
\]

\[
\text{initiates}(\text{stack\_on}(\text{robot}(1), X, Y), \text{on}(X, Y), T) \leftarrow \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(1), X), T), \\
\quad \text{holds\_at}(\text{top}(Y), T).
\]

\[
\text{initiates}(\text{stack\_on}(\text{robot}(1), X, Y), \text{top}(X), T) \leftarrow \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(1), X), T), \\
\quad \text{holds\_at}(\text{top}(Y), T).
\]

\[
\text{initiates}(\text{stack\_on}(\text{robot}(1), X, Y), \text{empty\_hands}(\text{robot}(1)), T) \leftarrow \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(1), X), T), \\
\quad \text{holds\_at}(\text{top}(Y), T).
\]

\[
\text{terminates}(\text{stack\_on}(\text{robot}(1), X, Y), \text{holding}(\text{robot}(1), X), T).
\]

\[
\text{terminates}(\text{stack\_on}(\text{robot}(1), X, Y), \text{top}(Y), T).
\]

\[
\text{initiates}(\text{give}(\text{robot}(R), \text{robot}(1), X), \text{holding}(\text{robot}(1), X), T) \leftarrow \\
\quad \text{neighbor}(\text{robot}(1), \text{robot}(R)), \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(R), X), T), \\
\quad \text{holds\_at}(\text{empty\_hands}(\text{robot}(1)), T).
\]

\[
\text{initiates}(\text{give}(\text{robot}(R), \text{robot}(1), X), \text{empty\_hands}(\text{robot}(1)), T) \leftarrow \\
\quad \text{neighbor}(\text{robot}(1), \text{robot}(R)), \\
\quad \text{holds\_at}(\text{holding}(\text{robot}(R), X), T), \\
\quad \text{holds\_at}(\text{empty\_hands}(\text{robot}(1)), T).
\]

\[
\text{terminates}(\text{give}(\text{robot}(R), \text{robot}(1), X), \text{holding}(\text{robot}(R), X), T).
\]

\[
\text{terminates}(\text{give}(\text{robot}(R), \text{robot}(1), X), \text{empty\_hands}(\text{robot}(1)), T).
\]

Now if we ask Robot 2 to achieve a configuration of its local environment (i.e. Region 2) such that \text{holds\_at}(\text{on}(‘D’, ‘A’), T), \text{holds\_at}(\text{top}(‘D’), T), we would expect the robots (Robot 2 may ask other robots for help) to come up with a valid planning after collaborative reasoning. One possible plan could be:

\[
\text{pick\_up}(\text{robot}(1), ‘A’) \Rightarrow \text{give}(\text{robot}(1), \text{robot}(2), ‘A’) \Rightarrow \\
\text{put\_down}(\text{robot}(2), ‘A’) \Rightarrow \text{pick\_up}(\text{robot}(4), ‘C’) \Rightarrow \text{put\_down}(\text{robot}(4), ‘C’) \Rightarrow \\
\text{pick\_up}(\text{robot}(4), ‘D’) \Rightarrow \text{give}(\text{robot}(4), \text{robot}(2), ‘D’) \Rightarrow \\
\text{place\_at}(\text{robot}(2), ‘D’, ‘A’)
\]

In the above example, though robot 2 knows how to build the required stack, but it doesn’t know where to get A and D, so it can ask the other robots (its neighbors) to find and
get the objects for it. Fortunately, robot 1 can find a plan to get A and robot 4 can find a plan to get D. Finally, robot 2 will “ask” the two neighbors to pass it the objects so it can continue to find a whole plan.

Let’s formalise the planning concept:

**Definition 5.1** There is a group of agents (robots) in the system (environment). Each agent has $\Sigma$, the set of effects rules (e.g. initiates and terminates) of the actions that the agent can execute, and has $\delta_n$, set of agent beliefs (e.g. initially, happens or persistent facts).

Let $EC$ be the set of simplified event calculus rules and $\Sigma$ be a given goal state (e.g. set of holds at), planning is to find a cluster of agents $AS$ and a set of actions (i.e. happens) $\Delta$ such that:

$$EC \cup \bigcup_{n \in AS} \delta_n \cup \Delta \models \Sigma$$

To make this kind of robots collaborative planning possible, I proposed a simple distributed abductive event calculus planner. The design of it is inspired by the abductive event calculus planner proposed by Murray [22]. The next section explains the distributed abductive planner in details.

### 5.2.2 Planner as Meta-Interpreter

For the planner, the set of abducibles are happens/2 and before/2. before is a partial relation of time points. For example, if $t_1 < t_2$ then we have $\text{before}(t_1, t_2)$.

Similarly to the abductive meta-interpreter, the planner meta-interpreter has a global abductive derivation.

$$g\_abdemo(Gs, DeltaIn, DeltaOut, NsIn, NsOut, Depth, AsIn, AsOut, Fixed, DontAsk).$$

In the $g\_abdemo/10$ predicate, the first argument $Gs$ is a list of sub-goals that describe the goal state. $DeltaIn$ is the current $\Delta$ when $g\_abdemo$ is called. $DeltaOut$ is the final (output) $\Delta$ obtained after $g\_abdemo$ succeeds. Both $DeltaIn$ and $DeltaOut$ are pairs of (Hs, Bs), where Hs is a list of happens and Bs is a list of before. NsIn and NsOut are the lists of sub-goals we must “fail to solve/prove”. Depth is used to limit the search space for the plans. For the current meta-interpreter, it indicates the maximum number of happens allowed in $\Delta$. AsIn is current list (cluster) of agents that have contributed to $\Delta$ and AsOut is a list of agents in the cluster after $g\_abdemo$ succeeds. Fixed is flag. When Fixed is set to yes, then if the agent is stuck on solving a sub-goal, it is allowed to ask only those helpers in the current cluster. This is typically true during a global consistency check (explain later). When Fixed is set to no, then the agent is free to ask any helper it discovers for an unsolved sub-goal. This is typically true during the global abductive derivation. Finally, DontAsk is a list of agents and it is used for prevent “looping”, e.g. if agent A asks agent B and C to solve a goal $G$, then B should not ask A and C to solve $G$.

The $g\_abdemo/10$ for a general sub-goal is defined as:
g_abdemo([G|Gs], (HsIn, BsIn), (HsOut, BsOut),
    NsIn, NsOut, D, AsIn, AsOut, F, DontAsk) :-
    check_length(Hs, D),
    resolve(G, (HsIn, BsIn), (Hs1, Bs1), Gs1, Ns1),
    append(Ns1, NsIn, Ns2),
    g_check_neg_goals(Ns2, (Hs1, Bs1), (Hs2, Bs2), AsIn),
    append(Gs1, Gs, Gs2),
    g_abdemo(Gs2, (Hs2, Bs2), (HsOut, BsOut),
        Ns2, NsOut, D, AsIn, AsOut, F, [Self])

The \texttt{g_abdemo} starts with checking the length of the plan. If the plan reaches the maximum length, then the search of the current branch should terminate and fail. After checking the length of the plan, it will try to resolve the next sub-goal $G$ by calling the predicate \texttt{resolve/5}:

\begin{verbatim}
resolve(G, DeltaIn, DeltaOut, Gs, Ns)
\end{verbatim}

The first two arguments of \texttt{resolve} are the sub-goal and the current $\Delta$. The rest of arguments are outputs – the output $\Delta$, the list of new sub-goals and the list of goals that must be failed to prove all the time.

\begin{itemize}
    \item If $G$ is a \texttt{happens} or \texttt{before}, and it is already in the current $\Delta$, then $\Delta$ remains unchanged and $Gs = Ns = []$.
    \item If $G$ is a \texttt{happens} and it is not in the current $\Delta$, then skolemise it if necessary, and add it to the current $\Delta$. In this case, $Gs = Ns = []$.
    \item If $G$ is a \texttt{before} and it is not in the current $\Delta$, then skolemise it if necessary and check whether it is consistent with the current $\Delta$. If yes, then add it to the current $\Delta$ and $Gs = Ns = []$.
    \item If $G$ is a negative goal \texttt{not}(G'), then $\Delta$ remains unchanged, $Gs = []$ and $Ns = \{[G']\}$.
\end{itemize}

Let’s look at the \texttt{g_abdemo} again. After it resolves a sub-goal $G$, it needs to perform a global consistency check on the new list of inconsistent goals (i.e. $Ns2$). Once the the global consistency check succeeds and a new $\Delta$ is returned, \texttt{g_abdemo} will continue to find a plan for the new list of sub-goals plus the rest of the sub-goals.

However, if \texttt{g_abdemo} fails to resolve a sub-goal, it will try to “ask for” help with the following definition:

\begin{verbatim}
g_abdemo([G|Gs], (HsIn, BsIn), (HsOut, BsOut),
    NsIn, NsOut, D, AsIn, AsOut, Fixed, DontAsk) :-
    find_helpers(G, Fixed, DontAsk, AsIn, Helpers),
    Request = (...),
    recruit_broker(Helpers, Request), !,
    fetch_answer(Broker, G, Answer),
    Answer = ((Hs1, Bs1), Ns1, As1),
    g_abdemo(Gs, (Hs1, Bs1), (HsOut, BsOut),
        Ns1, NsOut, D, As1, AsOut, F, [Self]).
\end{verbatim}
This version of \texttt{g\_abdemo} does the following:

1. It looks up a list of \texttt{PotentialHelpers} either from the central directory or from the local directory. If \texttt{Fixed == yes}, then \texttt{Helper = (PotentialHelper – \texttt{DontAsk}) \cap AsIn}; otherwise, \texttt{Helper = PotentialHelper – \texttt{DontAsk}}.

2. It creates a request message to be sent to the helpers. If using system version 2, this will be a KQML message.

3. It asks a broker thread to send out the requests to the helpers and to buffer returning answers/sub-plans (See Section 4.2.2).

4. If an answer/sub-plan is found and returned, then continue to find a plan for the remaining sub-goals.

5. While backtrack to \texttt{fetch\_answer/3}, the next buffered answer/sub-plan will be returned. If there is no more buffered answer/sub-plan, then backtrack fails.

Now let’s look at the global consistency check:

\texttt{g\_check\_neg\_goals(Ns, (HsIn, BsIn), (HsOut, BsOut), As)}.

The global consistency check is similar to the global consistency derivation described in Section 5.1.2. \texttt{g\_check\_neg\_goals} takes a list of inconsistent goals, the current \(\Delta\) and the agent clusters as input argument. It will pass the inconsistent goals and \(\Delta\) around the agents. \(\Delta\) may expand after one round consistency check. If this is the case, another round of consistency check will be initiated again and again until \(\Delta\) becomes “stable”. and then the global consistency check will succeed and the “stable” \(\Delta\) will be returned.

During one consistency check round, when an agent receives the inconsistent goals and the input \(\Delta\), it will try to “fail” all the inconsistent goals by calling:

\texttt{g\_abdemo\_fail\_all([], Delta, Delta, As)}.

\texttt{g\_abdemo\_fail\_all([Gs|Gss], DeltaIn, DeltaOut, As)} :-
\quad g\_abdemo\_fail\_any(Gs, DeltaIn, D1, As),
\quad g\_abdemo\_fail\_all(Gss, D1, DeltaOut, As).

That is, for each inconsistent goal, the agent only needs to any of its sub-goals. The \texttt{g\_abdemo\_fail\_any} succeeds if and only if:

1. There is a sub-goal \texttt{before(X, Y)}, and \(X == Y\).
2. There is a sub-goal \texttt{before(X, Y)}, and \(\Delta \models \texttt{before(Y, X)}\).
3. There is a sub-goal \texttt{before(X, Y)} and \(\Delta \not\models \texttt{before(X, Y)}\), then \texttt{before(Y, X)} is added to \(\Delta\).
4. There is a sub-goal \texttt{happens(E, T)} and \(\texttt{happens(E, T)} \notin \Delta\).
5. There is a non-abducible sub-goal and it cannot be resolved.
6. There is a negative sub-goal \texttt{not(G)}, and \texttt{g\_abdemo} succeeds with \([G]\) as the goal, \(\Delta\) as the initial \(\Delta\), \texttt{Fixed} set to \texttt{yes}. 

Example Run of the Planner

The distributed planner has been implemented basing on the second version of the distributed reasoning system.

Let’s try out the example described in Section 5.2.1.

When the robots started up and joined the group, they broadcasted their advertisements. For example, robot 1 broadcasted the advertisement:

\[
\text{advertise([}
\begin{align*}
& \text{from(robot(1)),} \\
& \text{content([} \\
& \quad \text{holds\_at(holding(robot(1), _), _),} \\
& \quad \text{holds\_at(empty\_hands(robot(1)), _)} \\
& \text{])} \\
\end{align*}
\text{])}.
\]

When we asked robot 2 to find a plan for \text{holds\_at(top('B'), T)} with depth 5, it returned a plan of \text{As} = \{\text{robot(1), robot(2)}\} and \text{Δ} = (\text{Hs, Bs}) where:

\[
\text{Hs} = [\text{happens(pick\_up(robot(1), 'A'), r1\_t3)}, \\
\text{happens(put\_down(robot(1), 'A'), r1\_t2)}, \\
\text{happens(pick\_up(robot(1), 'B'), r1\_t1)}, \\
\text{happens(give(robot(1), robot(2), 'B'), r2\_t2)}, \\
\text{happens(put\_down(robot(2), 'B'), r2\_t1)}].
\]

\[
\text{Bs} = [\text{before(r1\_t3, r1\_t2)}, \\
\text{before(r1\_t2, r1\_t1)}, \\
\text{before(r1\_t1, r2\_t2)}, \\
\text{before(r2\_t2, r2\_t1)}].
\]

If we topologically sort the actions, we will have a plan:

\[
\text{pick\_up(robot(1), 'A')} \Rightarrow \text{put\_down(robot(1), 'A')} \Rightarrow \text{pick\_up(robot(1), 'B')} \\
\Rightarrow \text{give(robot(1), robot(2), 'B')} \Rightarrow \text{put\_down(robot(2), 'B')}
\]

The trace of the planning is:

1. robot 2 picked the rule \text{initiates(pick\_up(robot(1), 'B'), top('B'), T)} but later failed because it couldn’t find a plan with length less than or equal to the depth.

2. robot 2 picked the rule \text{initiates(put\_down(robot(1), 'B'), top('B'), T)} and had \text{holds\_at(holding(robot(1), 'B'), T)} as the new sub-goal. Since it didn’t see \text{B} initially, there was only candidate rule – \text{initiates(give(robot(R), robot(2), 'B'), holding(robot(2), 'B'), T)}. So it picked that rule. At the first attempt, it resolved \text{neighbor(robot(2), robot(1))} and it had \text{holds\_at(holding(robot(1), 'B'), T)} as one of the sub-goals. Robot 2 couldn’t solve it but robot 1 had made such an advertisement. So robot 2 asked robot 1 to solve the goal. Luckily, robot 1 found a sub-plan for that goal and it passed the global consistency check between the two robots.
3. robot 2 received the sub-plan from robot 1 and had only holds_at(empty_hands(robot(r2)), T) as its last sub-goal. The last sub-goal was easily solved and hence the plan was returned.

4. backtracking, etc.

In fact, another plan returned by robot 2 is As = [robot(1), robot(2)] and the sorted actions are:

\[
pick\_up(\text{robot}(1), 'A') \Rightarrow put\_down(\text{robot}(1), 'A') \Rightarrow pick\_up(\text{robot}(1), 'B') \Rightarrow give(\text{robot}(1), \text{robot}(2), 'B') \Rightarrow stack\_on(\text{robot}(2), 'F')
\]

5.2.3 Optimisations

While implementing the distributed planner, quite a few optimisations are possible. The most important optimisation is to reduce the communications during the global consistency checks. In fact, for some applications like the blocks world we looked at, we can observe that the actions of each robot such as pick\_up and put\_down are independent. For example, robot 1 picks up object X in region 1 will not cause any conflict while robot 2 is putting down object Y in region 2. In this case, each robot doesn’t have to perform a global consistency check after it adds an abducible action to the ∆. It can perform a local consistency check instead. However, for other applications such as two robots working in the same region, then the global consistency checks are unavoidable.

The second possible optimisation is to maintain a transitive closure of the before relations in ∆, i.e. every time when before(T1, T2) is added to Bs of ∆, recompute Bs to get the transitive closure. Whenever we want to for consistency of a before and ∆, we just need to check whether Bs of ∆ contains it or not. Maintaining the transitive closure may also help to speed up global consistent check. For example, if we don’t maintain the closure, then during a round of consistency check if an agent adds a consistent before(T1, T2) to the ∆, we will have to start another round of consistency check. However, the downside of keeping the closure is that it will increase the size of ∆ and hence increase the payload of a request message when an agent is asking for help (because ∆ needs to be sent with the goal).

The third possible optimisation is to modify the way the planner abduces befores. We may abduce a set of happens first, and then pass it with the action effect rules to a constraint solver and find out the order between the actions. This sounds promising but needs to be further investigated.
Chapter 6

Porting to the Gumstix

On of the ultimate objectives of my project is to run abduction on the gumstix computers. To be more ambitious, we want to see if the DARES system can be run on the gumstix computers too. To achieve that, I have ported the YAP Prolog and QuProlog to ARM Linux so they can be run on the gumstix computers. I have tested the re-engineered meta-interpreter (See Chapter 3) with YAP Prolog on a gumstix. I have also run DARES with QuProlog 8.0 on the gumstix. This chapter describes the porting process and some test results for running abduction on the gumstix.

6.1 Cross Compiling the Prolog Engines

“Why was cross-compilation (porting) necessary?"

This is because the meta-interpreters and DARES are implemented in Prolog and they must be interpreted or executed by a Prolog engine. A gumstix and a lab machine have different architectures. A gumstix is running an ARM-ported Linux as its OS. The kernel (as well as some system utilities) of ARM-ported Linux are rewritten for a small and harsh runtime environment (e.g. slow processor, small memory and limited storage). Therefore, the Prolog engines compiled for a lab machine will not run on a gumstix and the cross-compilation of them is necessary (See Figure 6.1).

Figure 6.1: Ported Prolog Engines for DARS

84
In fact, since DARES is written in Prolog and is platform independent, as long as we can port the Prolog engine for different platforms, we will be able to run DARS on different devices (e.g. mobile phones, PDA, etc.) and each DARES agent can communicate with each other via wired or wireless network connection (See Figure 6.2 for an example). For this project, I have successfully created such an agent community with two gumstix and multiple lab PCs.

![DARES with Different Devices](image)

Figure 6.2: DARES with Different Devices

### 6.1.1 Selecting the Suitable Prolog Engines

Since there are various Prolog engines available, choosing a suitable one to port was the first important task. I have considered the following criteria while evaluating different well-known Prolog engines:

- **License requirement** – Some Prolog engines (e.g. BinProlog\(^1\)) are claimed to be very efficient but they are not free. So I can’t port it for this project without the authors’ permissions.

- **Availability and quality of the source code** – Some prolog engines (e.g. GNU Prolog\(^2\)) are not supported anymore, though the source code is still available. They are not preferred. Some less popular Prolog may be still under development but the code is quite buggy or the build system is too complicated to perform cross compilation. They are not preferred, either.

- **Efficiency and size** – We want to port and run the Prolog engine on a gumstix. So small yet efficient engines are preferred.

---

\(^1\)[BinProlog](http://www.binnetcorp.com/BinProlog/)

\(^2\)[GNU Prolog](http://www.gprolog.org)
• *Special Built-in Facilities* – For example, multi-threads support and network communication support.

I have chosen and ported two Prolog engines during my project. The first selected Prolog engine is YAP (Yet Another Prolog) and it has been used for benchmarking my re-engineered abductive meta-interpreter during the investigation phase. YAP is developed at LIACC/Universidade and at COPPE Sistemas/UFRJ. YAP prolog was chosen because it is free, open source and under GNU license. It is also known to be the most efficient among the available free engines. Its speed is claimed to be comparable with Sicstus Prolog. However, its size is approximately 4.77MB and it lacks support from its developers. The second selected Prolog engine is QuProlog and it is the one DARES was implemented with. QuProlog was selected because of the following:

• It is open source and free.
• It provides multi-threads support and has simple agent communications mechanism.
• The author (Peter Robinson) is available and can provide lots of vital supports (e.g. fixing bugs, answering questions).

During the project, I have been given two versions of QuProlog (v7.4 and v8.0). The two versions have major differences and below is a comparison table between them. Both of the versions have been ported but I have tested v8.0 the most.

<table>
<thead>
<tr>
<th>Version 7.4</th>
<th>Version 8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Peer-to-Peer communications via ICM. ICM is stable and its source is available. However, ICM is no longer supported by its developers.</td>
<td>• Peer-to-Peer communications via Pedro (i.e. ICM and Elvin are no longer required). However, the Pedro package is still at experimental stage and hence contains quite a few bugs.</td>
</tr>
<tr>
<td>• Publish/Subscribe communications via Elvin. Unfortunately, Elvin’s source is no longer available.</td>
<td>• This version is not officially released yet and hence is relatively less reliable.</td>
</tr>
<tr>
<td>• However, this version is relatively more stable as it has already been for a while in the lab.</td>
<td></td>
</tr>
</tbody>
</table>

### 6.1.2 The Cross-Compilation Process

A gumstix has only 16MB for storing the filesystem. Therefore it is impossible for its system to have various compilers and build tools installed. If we want to compile a package for the gumstix, we need to install all the necessary ARM Linux compilation tools to an ordinary Linux lab machine (called the *build* machine), compile the package on the build machine and move the binaries to the gumstix (called the *host* machine). Finally, the binaries of the package may be run on the gumstix. This process is called *cross-compilation*. Figure 6.3 shows a very simple case of it.

---

3YAP, [http://www.ncc.up.pt/~vsc/Yap/](http://www.ncc.up.pt/~vsc/Yap/)
4LIACC, [http://www.ncc.up.pt/liacc](http://www.ncc.up.pt/liacc)
5COPPE, [http://www.cos.ufrj.br/](http://www.cos.ufrj.br/)
Unfortunately, the cross-compilation processes for YAP and QuProlog were not as easy as you might expect due to the following problems:

- QuProlog is written in C++ and the standard C++ libraries (STL) was not installed on the gumstix by default. In this case, I had to re-compile the kernel and “reflash” the file system (explain later).

- QuProlog depends on some other packages (e.g. ICM, Elvin, Pedro, etc.), and some of them depend on some other packages and so on. In this case, I had to compile all the depended packages first before continue the process.

- Some kernel API used by QuProlog are not available or are different in ARM Linux. In this case, I had to “hack” the source code.

- During the build process, the QuProlog configuration tool needs to compile and run some test programs to determine some architecture dependent parameters. To cope with this, I extracted those test programs from the configure file, cross-compiled them and ran them on the gumstix to find out the correct values of the parameters.

- During the build processes of YAP and QuProlog, binaries produced need to be run in order to generate some non-binary output. Since the cross-compiled binaries couldn’t be run on the build machine (it is for run on the host machine, i.e. the gumstix), I had to do a “parallel compilation” trick (explained later).

**Refloshing the Filesystem**

If the package to be cross-compiled requires some kernel modules or libraries that are not currently installed, usually we need to re-configure and re-(cross) compile the the kernel and
the filesystem. Again, this is done on the build machine. A new filesystem will be created after the cross-compilation. We then need to “upload” the filesystem image the ROM (16MB) of the gumstix via serial port. The previous filesystem will be erased. Figure 6.4 illustrates the process.

![Figure 6.4: Reflashing the Filesystem for Gumstix](image)

Parallel Compilation

*Parallel Compilation* is needed when a cross-compilation needs to execute the generated binaries during the build process. For example, after the YAP compiler binaries are produced, the build process needs to execute it to compile some default Prolog libraries for the engine. Also, after the engine binaries are produced, the build process will need to run the engine once to generate some necessary bootstrap files. Since the build machine doesn’t understand these foreign binaries, it cannot execute it in anyway and the build process will fail. One workaround (See Figure 6.5) for this is to compile a normal version of YAP (same as QuProlog) for the build machine first. When the build process fails to execute the foreign binaries and stops, we can manually execute the native binaries to generate those necessary non-binary files and resume the build process. At the end, we copy all the foreign binaries and the non-binaries to the host machine. However, this trick does not always work!

6.1.3 Network Setup

Before running DARES on the gumstix computers, we need to connect them to a network. For testing, I have used the Department of Computing departmental network. Lab machines running as DARES agents have already been wired into the network. The two gumstix are connected to the wireless network and are given the names (*gumstix2* and *gumstix3*) and fixed addresses (*146.169.25.60* and *146.169.25.59*). While testing the second version of DARES, it is recommended to run the Pedro server on a reliable machine with fast network connection. I have run Pedro on *shell1* for the testing.
6.2 Testing and Benchmarking

Running the Abductive Meta-Interpreter on Gumstix

The re-engineered meta-interpreter has been run on the Gumstix with the YAP Prolog. I have tested it with many small logic programs and it worked very well. Besides, I did a benchmarking for it using the huge generated logic program described in Section 3.1.2. Below is the performance result:

<table>
<thead>
<tr>
<th>Queries</th>
<th>CPU Time (seconds)</th>
<th>Max. Stack (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0(test)</td>
<td>57.73</td>
<td>1220608</td>
</tr>
<tr>
<td>a1(test)</td>
<td>55.63</td>
<td>1220608</td>
</tr>
<tr>
<td>a2(test)</td>
<td>55.31</td>
<td>1220608</td>
</tr>
</tbody>
</table>

As you can see, the meta-interpreter running on the gumstix is approximately 25 times slower than running on an ordinary PC (in this case). However, this is expected because the slow processing speed of the gumstix. But the good thing is that the memory required for the stack is more or less the same as that running on an ordinary PC.

After other testings with small logic programs (e.g. < 1000 predicates), it is shown the meta-interpreter can run without being notably slow (about 3 times slower).
In conclusion, it is practical to run abduction on the Gumstix computers with the re-engineered abductive meta-interpreter.

Running DARES on Gumstix

The DARES system version 2 has also been tested on the Gumstix with QuProlog 8.0. I tested the general purpose distributed abductive meta-interpreter with small logic programs and they all gave correct results. I also tested the simple abductive planner with the blocks world example and with very simple queries. However, since there are several bugs in Pedro (e.g. it couldn’t handle message with size greater than 1KB), I haven’t done a quantitative evaluation with DARES using larger examples or queries. A quantitative benchmarking of DARES will be conducted as soon as the Pedro bugs are fixed.
Chapter 7

Conclusion

In the end of this project, I have achieved the initial objectives. The experiences I gained and lessons I learned are summarised in Section 7.1. As this is also an open ended project, many interesting ideas/findings are worth further investigation. There are also many possibilities for improving and extending DARES. The ideas and future developments are outlined in Section 7.2. Finally, Section 7.3 briefly talks about the challenges I faced during the project and how I would tackle them differently if I were to do the project again.

7.1 Summary

This project can be divided into three phases.

In the first phases, I have focused on the initial investigation of aduction. Re-engineering the abductive meta-interpreter (Section 3.1.1) by Kakas has helped me to understand the Kakas-Mancarella proof procedure in depth. The re-engineered meta-interpreter is notably fast comparing to the original one so it has been integrated as part of the general purpose distributed abductive meta-interpreter (Section 5.1.2). During the investigation, I have also tried out the idea of using partial evaluation to speed up the reasoning process (Section 3.2). A partial evaluator has been written to transform the logic programs into self-abductive logic programs. The initial results verify that partial evaluation can indeed help to speed up abductive reasoning with (ground) programs. However, due the to huge size of the transformed logic program, this optimisation is not particularly suitable for use on gumstix or other devices.

In the second phase, I have identified the basic assumptions (Section 4.1.3) and the desired properties (Section 4.1.4) of a distributed reasoning system. I have designed and implemented the first prototype (Section ??) of the system. In this version, the system consists of a central directory server and a group of reasoning agents. The reasoning agents are multi-threaded and can find their helpers when needed via the central server. The system has evolved to version 2 (Section ??) later in the project. The new version removes the central directory server. Each agent maintains a local directory and hence is able to look up potential helpers locally. This change reduces the number of message sent over the network during agents collaborations and therefore is more preferred for running on small devices. Evaluations between the two versions also show that the adoption of KQML message format will reduce the efficiency of agent communications, though on the other hand it can provide generic-ness.

In the third phase, I have proposed a general purpose meta-interpreter and a simple event calculus planner (Section 5.2). I have also outlined the proof of correctness for former
in Section 5.1.2. The multi-agent system together with the meta-interpreters finally give the Distributed Abductive REasoning System (DARES). DARES solves the issues described in Chapter 1 by:

- allowing a agent to discover helpers when it needs help.
- forcing global consistency check between the agents during collaborations and hence guarantees that the returned result is consistent among the agents.

As for the ambitious goal, DARES has been run and tested on both ordinary lab PCs and the gumstix computers. That is, the gumstix computers can act like intelligent agents and can perform (simple) abductive reasoning tasks collaboratively. My practical experiences of porting YAP Prolog and QuProlog have been described in Chapter 6.

7.2 Further Work

During this project, I have touched quite a few new ideas that I haven’t had time to further investigate. In addition, after testing the system on the actual gumstix computers, I learn better about the system’s performance and I come up with new ideas to further improve it. The following parts of this section summarises the extension work and issues that I wish to do or investigate in the future.

7.2.1 Optimisations and Extensions of DARES

There are a few possible optimisations that may push DARES to version 3.

- First of all, I would like to implement a graphical user interface for human user to interact with the agents in the system. The user interface can also be used for system debug.

- We have learned that the KQML will increase the message payload quite significantly. In the next version of DARES, I would like to look at some other better industrial standards or to formalise a new standard of message format.

- For the current version of the system, new threads are created for each incoming request and will be destroyed after the reasoning task is finished. The creation and destruction of threads will incur overhead. One possible improvement is to maintain a worker pool and allocate idle threads for incoming requests.

Since DARES is designed as a generic framework for cooperative reasoning, we may adapt other reasoning procedure to it:

- There are other well known proof procedure existing for abductive reasoning. I would like to investigate how they can be “port”ed to DARES.

- I also would like to investigate how the existing (and future) meta-interpreters can be enhanced or optimised. For example, the abductive planner I have proposed is for demonstration of the collaborative planning with DARES. To apply it to real world applications, it needs to be modified to work with the full version of event calculus.


7.2.2 Identify Potential Applications

For this project, DARES has been developed as a prototype. There are two identified potential applications for it:

- The use in sensor networks. Each sensor can receive monitoring data and represent it in declarative form. The sensors may reason collaboratively from the data to find explanations for the current reading, and then decide a suitable action for some certain (abnormal) situations.

- Planning in robotics. We can use the gumstix computers as robot controllers and run DARES on them. This will allow the robots to reasoning together on their perception, and will also allow them perform planning together.

In the future, I hope to investigate the above applications as well as to find more suitable applications for DARES.

7.3 Final Thoughts

This is a research and development project. It requires both the work of theoretical A.I. and Practical A.I.

During the project, some of the main challenges I have faced are:

- There wasn’t much help about using the Gumstix computers. I had to find out how to make them work myself. When I encountered problems, the only place I could hope to get help from was the user group. However, many times after I posted the questions, no one seemed to be able to or bother to answer them.

- In order to get the most from QuProlog, I used its latest (but not yet released) version 8.0 for implementing DARES. There were several bugs in both QuProlog and Pedro. Sometimes it took me hours or days to debug my system, but at the end it turned out to be the problem of Pedro (e.g. couldn’t parse some Prolog terms correctly or couldn’t send big messages of the size greater than 1K). But fortunately, the author Robinson was very helpful and helped me to fixed most of the bugs promptly.

- Lots of work but not enough time. Many new ideas and objectives were added along the course of the project.

I think the project has been successful and I have delivered the expected outputs. However, if I were to do the project again, I would try to limit the scope of it and focus on one task at a time. Also, I would try to avoid using un-tested software and hardware to minimise the time wasted on fixing problems unrelated to the project’s main goals.
Bibliography

[1] UMBC Agent Web – KQML.


Appendix A

Sample Code

A.1 Kakas ALP Meta-Interpreter

```prolog
/* ================================================= =============== */
/* ============== AN ABDUCTIVE PROOF PROCEDURE ========= =============== */
/* An abductive program is a triple <P,A,IC> where P is a normal */
/* logic program, A is a set of abducible predicates and */
/* IC is a set of integrity constraints. */
/* Predicate symbols are declared as abducibles using the */
/* predicate: "abducible_predicate" (eg. abducible_predicate(p). ) */
/* Abducibles must have no rules in the program. */
/* Integrity Constraints must be in the form of a denial with at */
/* least one abducible condition, and must be written as rules */
/* in the program in the form: */
/* "ic:- conjunction of positive or negative literals." */
/* All abducibles in the program must be ground at the time of call. */
/* This procedure deals with NAF through Abduction. */
/* Negative conditions are written as not(p). */
/* The semantics of NAF computed is that of partial stable models */
/* (or equivalently preferred extensions). */
/* This abductive proof procedure runs in SICStus Prolog. */
/* A query is posed as: demo(List_of_Goals,[],Output_Variables). */
/* ============================================================== */
/* ABDUCTIVE PHASE */
/* ============================================================== */
demo([],Exp,Exp).
demo(Goals,ExpSoFar,FinalExp) :-
    select_literal(Goals,SelectedLiteral,RestOfGoals),
    demo_one(SelectedLiteral,ExpSoFar,InterExp),
    demo(RestOfGoals,InterExp,FinalExp).
/* ============================================================== */
```
APPENDIX A. SAMPLE CODE

/* Positive non-abducible */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_positive(SelectedLiteral),
    \+ abducible(SelectedLiteral),
    clause_list(SelectedLiteral, Body),
    demo(Body, ExpSoFar, InterExp).

/* Positive Ground abducible which is already assumed */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_positive(SelectedLiteral),
    abducible(SelectedLiteral),
    ground(SelectedLiteral),
    in(SelectedLiteral, ExpSoFar),
    InterExp=ExpSoFar.

/* Negative Ground abducible or non-abducible which is already assumed */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_negative(SelectedLiteral),
    complement(SelectedLiteral, Complement),
    ground(Complement),
    in(SelectedLiteral, ExpSoFar),
    InterExp=ExpSoFar.

/* Negative Ground non-abducible */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_negative(SelectedLiteral),
    complement(SelectedLiteral, Complement),
    ground(Complement),
    \+ abducible(Complement),
    add_hypothesis(SelectedLiteral, ExpSoFar, InterExpSoFar),
    demo_fail_ICS(SelectedLiteral, InterExpSoFar, InterExp).

/* Negative Ground abducible which is not assumed */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_negative(SelectedLiteral),
    complement(SelectedLiteral, Complement),
    ground(Complement),
    \+ in(Complement, ExpSoFar),
    add_hypothesis(SelectedLiteral, ExpSoFar, InterExp).

/* Positive Ground abducible which is not assumed */
demo_one(SelectedLiteral, ExpSoFar, InterExp) :-
    is_positive(SelectedLiteral),
    abducible(SelectedLiteral),
    ground(SelectedLiteral),
    \+ in(SelectedLiteral, ExpSoFar),
    complement(SelectedLiteral, Complement),
    \+ in(Complement, ExpSoFar),
    add_hypothesis(SelectedLiteral, ExpSoFar, InterExpSoFar),
    demo_fail_ICS(SelectedLiteral, InterExpSoFar, InterExp).
/* CONSISTENCY PHASE */

demo_fail_ICS (Abducible,HypSoFar,NewHypSoFar) :-
    findall (IntConDenial, 
        clause_list(ic,IntConDenial), 
       ListOfIntConDenials), 
    findall (OneResolventDenial, 
        (in(OneConDenial,ListOfIntConDenials), 
        resolve(Abducible,OneConDenial,OneResolventDenial)), 
        ListOfResolventDenials), 
    demo_failure (ListOfResolventDenials,HypSoFar,NewHypSoFar).

demo_failure([],HypSoFar,HypSoFar).

demo_failure (ListOfResolventDenials,HypSoFar,NewHypSoFar) :- 
    select_first_denial(ListOfResolventDenials,SelectedDenial,RestOfDenials), 
    
    demo_failure_leaf(SelectedDenial,HypSoFar,InterHyp),
    demo_failure(RestOfDenials,InterHyp,NewHypSoFar).

demo_failure_leaf(SelectedDenial,HypSoFar,InterHyp) :- 
    select(SelectedDenial,SelectedLiteral,RestOfLiterals),
    demo_failure_on_literal(SelectedLiteral,RestOfLiterals, 
        HypSoFar,InterHyp).

demo_failure_on_literal(SelectedLiteral,RestOfLiterals, 
        HypSoFar,InterHyp) :- 
    is_positive(SelectedLiteral), 
    
    demo_failure (ListOfNewConjunctions,HypSoFar,InterHyp).

demo_failure_on_literal(SelectedLiteral,RestOfLiterals, 
        HypSoFar,InterHyp) :- 
    is_positive(SelectedLiteral), 
    abducible(SelectedLiteral),
    ground(SelectedLiteral),
    in(SelectedLiteral,HypSoFar),
    demo_failure ([RestOfLiterals],HypSoFar,InterHyp).

demo_failure_on_literal(SelectedLiteral,RestOfLiterals, 
        HypSoFar,InterHyp) :- 
    is_positive(SelectedLiteral), 
    abducible(SelectedLiteral),
    ground(SelectedLiteral),
    in(SelectedLiteral,HypSoFar),
    demo_failure ([RestOfLiterals],HypSoFar,InterHyp).
demo_failure_on_literal(SelectedLiteral, RestOfLiterals, HypSoFar, InterHyp) :-
   is_negative(SelectedLiteral),
   complement(SelectedLiteral, Complement),
   ground(Complement),
   in(SelectedLiteral, HypSoFar),
   demo_failure([[RestOfLiterals], HypSoFar, InterHyp]).

/* Positive Ground abducible which is not assumed */
demo_failure_on_literal(SelectedLiteral, RestOfLiterals, HypSoFar, InterHyp) :-
   is_positive(SelectedLiteral),
   abducible(SelectedLiteral),
   ground(SelectedLiteral),
   \+ in(SelectedLiteral, HypSoFar),
   complement(SelectedLiteral, Complement),
   add_hypothesis(Complement, HypSoFar, InterHyp).

/* Negative Ground abducible or non-abducible */
demo_failure_on_literal(SelectedLiteral, RestOfLiterals, HypSoFar, InterHyp) :-
   is_negative(SelectedLiteral),
   complement(SelectedLiteral, Complement),
   ground(Complement),
   \+ in(SelectedLiteral, HypSoFar),
   demo([[Complement], HypSoFar, InterHyp]).

/* ============================================================== */
/* LOW-LEVEL PREDICATES */
/* ============================================================== */
in(X, [X|Y]).
in(X, [Z|Y]): - in(X,Y).
concat([], L, L).
concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
select([X|Y], X, Y).
select([Z|Y], X, [Z|Rest]) :- select(Y, X, Rest).
select_literal([[SelectedLiteral|RestOfLiterals], SelectedLiteral, RestOfLiterals]).
select_first_denial([[FirstDenial|RestDenials], FirstDenial, RestDenials]).
abducible(Atom) :- get_predicate_name(Atom, PredicateName),
   abducible_predicate(PredicateName).
get_predicate_name(Atom, PredicateName) :-
   (Atom) =.. [PredicateName | RestOfAtom].
clause_list(Head, []) :-
   prolog:current_predicate(_, Head),!,
APPENDIX A. SAMPLE CODE

Listing A.1: Kakas' Meta-Interpreter

A.2 The Distributed Family Tree Database
11  rule(parent(pat, jim), []).
12
13  rule(male(tom), []).
14  rule(male(bob), []).
15  rule(male(jim), []).
16
17  rule(father(X, Y),
18      [male(X), parent(X, Y)]).
19
20  rule(grandparent(X, Z),
21      [parent(X, Y), parent(Y, Z)]).
22
23  rule(predecessor(X, Z),
24      [parent(X, Y), predecessor(Y, Z)]).
25
26  % ==== Agent 2 ======
27
28  knows(parent(_, _)).
29  knows(male(_)).
30  knows(offspring(_, _)).
31  knows(father(_)).
32  knows(predecessor(_, _)).
33
34  rule(parent(bob, ann), []).
35  rule(parent(bob, pat), []).
36  rule(parent(pat, jim), []).
37
38  rule(male(tom), []).
39  rule(male(bob), []).
40  rule(male(jim), []).
41
42  rule(offspring(Y, X),
43      [parent(X, Y)]).
44
45  rule(father(X, Y),
46      [male(X), parent(X, Y)]).
47
48  rule(predecessor(X, Z),
49      [parent(X, Y), predecessor(Y, Z)]).

Listing A.2: Sample Database for the Distributed Deductive Meta-Interpreter