2012

Integrating optimization and preference handling in agent programming frameworks

Aniruddha Dasgupta
University of Wollongong

Recommended Citation
UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
INTEGRATING OPTIMIZATION AND PREFERENCE HANDLING IN AGENT PROGRAMMING FRAMEWORKS

A Dissertation Submitted in Fulfilment of the Requirements for the Award of the Degree of

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Aniruddha Dasgupta

School of Computer Science and Software Engineering
Faculty of Informatics
CERTIFICATION

I, Aniruddha Dasgupta, declare that this thesis, submitted in fulfilment of the require-
ments for the award of Doctor of Philosophy, in the School of Computer Science and
Software Engineering, Faculty of Informatics, University of Wollongong, is wholly my
own work unless otherwise referenced or acknowledged. The document has not been
submitted for qualifications at any other academic institution.

Aniruddha Dasgupta
October 4, 2012
Dedicated to

My Wife Susmita
and
my sons Amrithesh and Shivam
# Table of Contents

List of Tables .................................................................................. v
List of Figures/Illustrations ................................................................. vi
Abstract ......................................................................................... vii
Publications arising from this thesis ..................................................... viii
Acknowledgements ........................................................................ x

## 1 Introduction

1.1 Motivation ................................................................................. 1
1.2 Overview and Highlights of the thesis ......................................... 3
1.3 Thesis Structure ........................................................................ 4

## 2 Background

2.1 Agents ..................................................................................... 6
   2.1.1 Properties of Agents .......................................................... 8
   2.1.2 Agent Architecture ......................................................... 9
   2.1.3 BDI-Based Languages and Platforms ................................. 16
   2.1.4 AgentSpeak(L) ................................................................. 19
2.2 Structured Operational Semantics ............................................ 24
2.3 Constraint Programming ......................................................... 26
   2.3.1 Constraint Satisfaction and Optimisation Problem ................. 26
   2.3.2 Constraint Logic Programming(CLP) ................................ 27
   2.3.3 CLP toolkits .................................................................. 30
   2.3.4 Soft Constraints ............................................................... 32
# TABLE OF CONTENTS

## 3 Agent Programming with CASO 36

3.1 Motivating example ................. 36  
3.2 Overview of CASO ................. 37  
  3.2.1 Belief Base ($\beta$) ....... 39  
  3.2.2 Set of plans ($P$) .......... 39  
  3.2.3 Set of Objective Functions ($\Theta$) .... 40  
  3.2.4 Set of events ($E$) ....... 41  
  3.2.5 Objective Store ($OS$) ....... 42  
  3.2.6 Set of intentions ($I$) ..... 44  
  3.2.7 Look ahead parameter ($n$) .... 44  
  3.2.8 Event Selection Function ($SE$) .... 45  
  3.2.9 Option Selection Function ($SO$) .... 45  
  3.2.10 Intention Selection Function ($SI$) .... 48  
3.3 Difference with Agentspeak(L) .... 49  
3.4 Operational Semantics ............ 50  
3.5 Related Work .................. 53  

## 4 Implementation details 56

4.1 CASO Interpreter ................. 56  
  4.1.1 Algorithm for CASO interpreter .... 58  
4.2 Integrating Jason ................. 59  
  4.2.1 Customizing Jason .......... 60  
4.3 Integrating ECLiPSe .......... 61  
  4.3.1 CASO execution cycle using CLP files .... 61  
4.4 Experimental Results ............ 64
TABLE OF CONTENTS

5 Extending CASO with User Preferences
  5.1 Introduction ................................................. 71
  5.2 BAOP: Extending CASO ...................................... 73
    5.2.1 Set of plans (P) ....................................... 73
    5.2.2 Set of user preferences (µ) ......................... 74
    5.2.3 Set of events (E) ...................................... 75
    5.2.4 Preference Store (PS) ................................. 76
    5.2.5 Option Selection Function (SO) ...................... 77
    5.2.6 Intention Selection Function (SI) ................. 81
  5.3 BAOP interpreter ........................................... 82
  5.4 Related Work .............................................. 84

6 CASO Case Study: Decision support in a biomass supply chain
  6.1 Biomass Supply Chain ....................................... 87
  6.2 Decision Making and Optimisation in BSC .................... 89
    6.2.1 Harvesting and Handling of Biomass ................. 90
    6.2.2 Optimisation considerations ......................... 93
  6.3 CASO agent in Biomass Supply Chain ....................... 93
    6.3.1 Stage 1: Cutting and Transporting Biomass to Storage . 94
    6.3.2 Stage 2: Transporting Biomass from Storage to Plant .... 99
    6.3.3 Stage 3: Storage of Biomass at Conversion Plant ...... 103
    6.3.4 Plan Selection at Stage 1 using 1 and 2-step lookahead .... 103

7 Constraint based Optimizations in other BDI languages
  7.1 Introduction ................................................. 105
  7.2 3APL .......................................................... 106
    7.2.1 3APL deliberation cycle .............................. 107
    7.2.2 Example ................................................. 108
    7.2.3 Extending 3APL with constraint-based optimization ... 110
  7.3 2APL .......................................................... 113
    7.3.1 Deliberation Cycle of 2APL ............................ 114
    7.3.2 Extending 2APL with constraint-based optimization ... 114
  7.4 CANPLAN ..................................................... 115
    7.4.1 Deliberation Cycle ...................................... 116
    7.4.2 Extending CANPLAN with constraint-based optimization ... 116
TABLE OF CONTENTS

8 Conclusion 119
   8.1 Summary and Discussion 119
   8.2 Future Work 120

Appendix 122

References 138
List of Tables

6.1 Rates for different types of trailers ........................................ 97
6.2 Rates for different types of trucks ........................................ 101
6.3 Possible paths ................................................................. 104
List of Figures

2.1 Agent and its environment ........................................ 7

3.1 Agent Plans and corresponding goal-plan tree .................. 47
3.2 Plan Selection with 1 step look-ahead ......................... 48

4.1 CASO execution cycle ............................................ 57
4.2 CASO Belief and Objective function written in ECLiPSe CLP style 62
4.3 Graphs showing results of Experiment 1 ......................... 66
4.4 Graphs showing results of Experiment 2 ......................... 67
4.5 Graphs showing results of Experiment 3 ......................... 67
4.6 Graphs showing results of Experiment 4 ......................... 68
4.7 Graphs showing results of Experiment 5 ......................... 69

5.1 Agent Plans and corresponding goal-plan tree ................. 78
5.2 BAOP execution cycle ........................................... 83

6.1 Plans related to top level goals of cutting and transporting to storage 96
6.2 Belief(CLP) of CASO Agent related to trailer fitted with grappler 98
6.3 Belief(CLP) of CASO Agent related to forestry trailer ......... 98
6.4 Belief(CLP) of CASO Agent related to flatbed trailer .......... 98
6.5 Plans related to transporting Biomass from Storage to Plant .... 100
6.6 Belief(CLP) of CASO Agent related to high sided truck ....... 102
6.7 Belief(CLP) of CASO Agent related to flat deck truck ......... 102
Abstract

There has been considerable amount of research in the recent past on multi-agent systems (MAS) by researchers in the area of Artificial Intelligence. The Belief-Desire-Intention (BDI) software model is a software model developed for programming intelligent agents. It uses the concepts of agent’s beliefs, desires and intentions to solve a particular problem in agent programming and provides a mechanism for separating the activity of selecting a plan from the execution of currently active plans. The BDI model provides a theoretical foundation for some of the essential features of autonomous agents and multi-agent systems.

For many applications, it is more convenient to let the user provide in real time, a more elaborate specification consisting of constraints and preferences over possible goal states. Then, let the system discover a plan for the most desirable among the feasible goal states. Although there has been considerable research in developing BDI based agent systems and languages, limited work has been done in developing BDI systems where users can specify explicit objectives and preferences in real time. In this thesis I provide extensions to BDI-based agent-oriented programming languages which help the agent in its decision making process. I integrate constraint logic programming into the BDI framework to provide the agent with enough autonomy to make measured and deliberative decisions in a constantly changing, dynamic environment. I further extend this framework to incorporate c-semiring based structure to capture user preferences.
List of publications

The major contributions of my thesis can be summarized as follows:

- Introduction of a new agent programming language called CASO based on the popular BDI [64] based language Agentspeak(L) [61] which incorporates user defined objectives into the agent reasoning cycle.

- Application of the CASO notion of objectives and optimization techniques to some other representative BDI languages

- Application of CASO in Biomass Supply Chain Management

- Extension of CASO to incorporate user defined preferences into the agent to create a new language called BAOP.

The following is a list of publications that have resulted from the research presented in the thesis.


Acknowledgements

First of all, I would like to thank to my supervisor, Prof. Aditya K. Ghose, for his careful guidance, friendship and support during my study years. I am always grateful to him for giving me the opportunity to enroll as a Ph.D. student at University of Wollongong.

I am especially thankful to my wife Susmita who has stood by me all along and always encouraged me for doing a Ph.D and without whose support this thesis would not have been possible. I wish to also thank my colleagues for their research inputs as well as my brother and my in-laws (especially Buku) for their continuous encouragement.

Finally, I would like to thank my parents without whose blessings and love I would not have reached so far in my life.
Chapter 1

Introduction

This chapter provides motivation for my research and also presents an outline of the structure of the thesis.

1.1 Motivation

Recent years have seen a rapid growth of interest in agent-oriented technology in several areas of computer science. Agents are being used for applications as diverse as electronic commerce, manufacturing, interface design, management of complex commercial systems, computer games, personalized information management and industrial processes. In the multi-agent systems (MAS) community, software agents are conceived as autonomous computational entities situated in some environments which they can sense and act upon in a dynamic (reactive and/or proactive) way according to the environment’s changes and their design objectives [83]. One of the research topics in MASs is the development of personal agents. These agents act on behalf of users, automating tasks that were previously performed by them. Each agent is given the mandate to achieve defined goals. To do this, it autonomously selects appropriate
actions, depending on the prevailing conditions in the environment, based on its own capabilities and means until it succeeds, fails, needs decisions or new instructions or is stopped by its owner. Thus decision agents can be designed to provide interactive decision aids for end-users by eliciting their goals and preferences and then recommending matching products.

Agent-oriented programming is highly suited for applications which are embedded in complex dynamic environments, and is based on human concepts, such as beliefs, goals and plans. This allows a natural specification of sophisticated software systems in terms that are similar to human understanding, thus permitting programmers to concentrate on the critical properties of the application rather than getting absorbed in the intricate detail of a complicated environment.

For most realistic applications an agent should have both reactive and deliberative behavior [22]. The first type of behavior addresses the recognition of emergency situations in time and provides rapid responses whereas the second type addresses the planning of actions to achieve its long term goals. Agent deliberation not only includes planning of the tasks, but also includes various types of decisions at each moment of time [38] such as how to select a goal from a set of possible goals and whether to behaving more reactively than deliberatively. These types of decisions, which constitute the agents deliberation process and determine the types of agent behavior, are usually implemented statically in an interpreter and are not explicitly and directly programmable in an agent module [21]. As an example, the decision on goal selection is usually based on an assumed predefined ordering on goals such that the interpreter always selects the goal which has the highest rank. At each moment of agent execution it should be decided which rules to select and apply and which goals to select and execute. These decisions are usually implemented in the interpreter and the deliberation choices are decided statically and thus hardwired in the interpreter. Since
these decisions are pre-programmed in the interpreter, it is quite difficult to program or influence the deliberation process of an agent.

1.2 Overview and Highlights of the thesis

The development of personal user agents has several challenges including the fact that objectives and user preferences as well as must be captured (objective and preference and elicitation) and modeled (objectives and preferences representation) in the system.

The major contribution of this thesis is the introduction of a new agent programming language based on the popular BDI \[64\] based language Agentspeak(L) \[61\] where I try to overcome the difficulties mentioned earlier. I introduce into an agent both deliberative and reactive features whereby objectives can be modeled. I call this new language CASO (Constraint AgentSpeak(L) with Objective) where users could give explicit objectives. This technique applies constraint and objective directed solving on the context section of a BDI agent’s plan specification in order to determine an applicable plan to fire. More interestingly, the use of constraint-based representations can make it possible to deal with explicit agent objectives (as distinct from agent goals) that express the things that an agent may seek to optimize at any given point in time.

I also give details of my implementation of CASO along with its formal operational semantics. For implementing CASO, I have used Jason \[9\], which is a popular and well known interpreter of AgentSpeak(L). I use ECLiPSe \[4\] as a Constraint Logic Programming \[54\] tool solver and incorporate it into the execution cycle of Jason.

Another contribution of this thesis is the incorporation of user preferences into the agent. For this I extend CASO to create a new language called BAOP(BDI Agent with Objectives and Preferences) and show how user preferences could also be modeled into the BDI execution cycle. My work here especially focuses on the use of
soft constraints in an agent environment where we give a quantitative dimension to this agent deliberation process by apply c-semiring \[7\] based techniques to determine the preferred solution.

One of the major outcomes of the techniques and methods that have been described here has been the application of CASO to a real life example of Biomass Supply Chain. This shows that CASO can be used in realtime application situations and could easily be extended to different domains. It is to be noted that my implementation of CASO follows a generic BDI structure and is not just confined to AgentSpeak(L). I also highlight this feature and give an overview of how other popular BDI languages could easily be extended with CASO like features.

Unlike most other work, I assume that the agent environment is highly reactive in nature and therefore constraints, preferences and objectives could change at every point in time. I give the user the ability to change objectives and preferences thereby making this a highly flexible system.

1.3 Thesis Structure

In chapter 2 I provide a background on agents along with the various agent based programming languages and platforms. The chapter also provides an overview of constraint satisfaction and optimization techniques which forms the basis for the rest of the thesis.

Chapter 3 introduces a new agent language CASO based on the agent programming language AgentSpeak(L). CASO augments the reactive agent with rational thinking by applying decision making techniques into the execution cycle of an agent.

The implementation details of CASO as well as some experimental results are given in chapter 4.
1.3. Thesis Structure

In chapter 5 I extend CASO to incorporate decision making strategies based on user preferences.

Chapter 6 describes how CASO can be used in real-life decision making where I take the example of a Biomass Supply Chain scenario.

In chapter 7 I discuss how the CASO notion of applying objectives and optimization techniques can be easily incorporated in some other representative agent based languages.

Finally, chapter 8 contains a summary of the work presented and also the work that would be attempted in the future.
Chapter 2

Background

This chapter provides a background on agents, constraints and optimization which form the backbone of this thesis.

2.1 Agents

There is no universally accepted definition of the term agent. Some define an agent as an entity that can be viewed as perceiving its environment through sensors and acting upon its environment through effectors. Some state that an agent is a hardware and/or software-based computer system displaying the properties of autonomy, social adeptness, reactivity, and pro-activeness. There is a consensus that autonomy, the ability to act without the intervention of humans or other systems, is a key feature of an agent. Beyond that, different attributes take on different importance based on the domain of the agent.

An agent is commonly seen as an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives [47]. An autonomous computing entity
2.1. Agents

capsulates its state and makes decisions about what to do based on this state, without the direct intervention of humans or others. The environment in which agents are situated can change during the execution of the agent. This requires flexibility in agent behavior, which means that the agent should be able to respond adequately to changes in its environment in such a way that it achieves its objectives or goals. "Intelligent" or rational agents are those agents that are able to make good decisions about what to do. Moreover the agents can negotiate or cooperate with other agents to achieve some common goal.

Figure 2.1 depicts a high-level view of an agent within its environment. An agent receives input from its environment and, through a repertoire of actions available to it, reacts to it in order to modify it. Generally in real world situations an agent will not have total control over its environment. Thus, the same action performed twice in identical situations may have completely different outcomes. Also the action taken by the agent may not produce the desired effect at all at times.

Some of the other properties of agents include mobility, veracity, benevolence, rationality and learning/adaptation.
2.1.1 Properties of Agents

One of the most important property of an intelligent agents is its autonomy. The autonomy of software agents is directly related to their capacity to make decisions without intervention of the human users. At the lowest level these decisions concern the next action that the agent is going to perform. However, the actions are selected based on many high level decisions such as whether it will stick to its current plan to reach a goal, whether it wants to change its goal, whether to adhere to a norm or not, etc. The various other properties of an intelligent agent \(^{[83]}\) that is capable of flexible autonomous action to meet its design objectives are described below.

**Social adeptness** Intelligent agents are capable of interacting with other agents (and possibly humans), through negotiation and/or cooperation, to satisfy their design objectives.

**Reactivity** Intelligent agents perceive and respond in a timely fashion to changes that occur in their environment in order to satisfy their design objectives. The agent’s goals and/or assumptions that form the basis for a procedure that is currently executing may be affected by a changed environment and a different set of actions may be need to be performed. A reactive agent is driven by a set of predefined actions for different situations and has no representation of its environment or of other agents and, therefore, is incapable of foreseeing what is going to happen. As a result, it is incapable of anticipating by planning what action to take. A reactive agent cannot carry out a priori reasoning. It merely reacts to the current situation, and its actions become part of the situation of the others. The actions that agents carry out would modify the agents environment and their future decision making.

**Pro-activeness** Reacting to an environment by mapping a stimulus into a set of
responses is not enough. As we want intelligent agents to do things for us, goal directed behavior is needed. In a changed environment, intelligent agents have to recognize opportunities and take the initiative if they are to produce meaningful results. The challenge to the agent designer is to integrate effectively goal-directed and reactive behavior.

**Mobility** The ability to move around an electronic environment.

**Veracity** An agent will not knowingly communicate false information.

**Benevolence** Agents do not have conflicting goals and every agent will therefore always try to do what is asked of it.

**Rationality** An agent will act in order to achieve its goals insofar as its beliefs permit.

**Learning/adaptation** Agents improve performance over time.

### 2.1.2 Agent Architecture

To create a language for programming agents which are capable of intelligent behaviour, researchers and developers have tried to address the following properties of agents.

- The basic constituent parts of an intelligent agent.
- The thinking and which deliberation strategy of an agent.
- The relationship between the agent’s beliefs and its goals?

Researchers have developed formal models and architectures of agents based on the above criteria. Maes [53] proposes agent architecture as a particular methodology for
building agents. It specifies how the agent can be decomposed into the construction of a ‘set of component modules and how these modules should be made to interact’. Agent architectures present a higher level of abstraction for building and viewing agent systems. Five agent architectural styles have been mentioned in the literature. They are reactive, planning, knowledge based, deliberative and Belief Desire Intention (BDI). The following sections review the most common types of agent architectures and the components they are constructed from.

2.1.2.1 Reactive Agent Architectures

A reactive architecture does not have a central world model and does not use complex reasoning [83]. Reactive agents act by stimulus-response to environmental states. The agent perceives an environmental change and reacts accordingly. Reactive agents can also react to messages from other agents. Although reactive agents are basic and can only perform simplistic tasks, they do form a building block from which other, more complex agents can be built. By adding a knowledge base to a simple reactive agent, the agent becomes capable of making decisions that take into account previously encountered state information. By adding goals and a planning mechanism a complex goal directed agent can be create. Although complex patterns of behavior can be developed using reactive agents, their primary goals usually consist of being robust and having a fast response time. Most agent architectures contain a reactive component of some kind although they may not be truly reactive agents. Majority of reactive architectures can be modeled using a basic IF-THEN rule structure.

2.1.2.2 Planning Agent Architecture

Planning is the process of formulating a list of actions in order to achieve a specified goal [60]. A planner uses knowledge about the actions it may perform and their
2.1. Agents

consequences. It uses this as well as knowledge about the environment, to formulate a list of acceptable state transforming operators that can transform the agent from an initial state into a goal state. Planning architectures are usually embedded in other agent architectures to determine the actions that an agent will perform. Within a given agent architecture, plans may be either synthesized dynamically or predefined in advance and placed in a plan library. The steps of a plan are generally operators containing parameters that need to be defined to a set value in order for the operator to function. A fully instantiated plan is one in which all of these parameters are defined to a set value. According to Russell and Norvig \cite{RN66} a plan is a formally defined data structure that contains the following components:

- A set of plan steps. Each step is one of the operators of the problem.
- A set of step ordering constraints.
- A set of variable binding constraints.
- A set of causal links to record the purpose(s) of steps in the plan

2.1.2.3 Knowledge-Based Agent Architectures

Knowledge-based systems use data structures consisting of explicitly represented problem-solving information. This knowledge can be viewed as a set of facts about the world. Three aspects of knowledge-based systems, which make them powerful, are:

- They can accept new tasks in the form of explicitly described goals.
- They can achieve competence quickly by being told or learning new knowledge about the environment.
• They can adapt to changes in the environment by updating the relevant knowledge.

In general, knowledge-based systems represent knowledge using a formal declarative language. The use of declarative language allows knowledge to be added or deleted from the knowledge base quickly and easily without affecting the rest of the system. Using a declarative language such as first-order logic also allows new information to be derived from the current knowledge stored in the system using inference mechanisms. An inference mechanism can perform two actions. First, given a knowledge base, it can generate new sentences that are necessarily true, given that the old sentences are true. Second, given a knowledge base and a sentence, it can determine whether the sentence was generated by the knowledge base or not [66].

2.1.2.4 Deliberative Agent Architectures

The deliberative agent architecture [83] contains an explicitly represented, symbolic model of the world. Decisions about what actions are to be performed are made via logical reasoning, based on pattern matching and symbolic manipulation. There are two important problems to be solved when building an agent in this way:

1. The transduction problem: that of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.

2. The representation/reasoning problem: that of how to symbolically represent information about complex real-world entities and processes, and how to get agents to reason with this information in time for the results to be useful.
2.1.2.5 BDI architecture

The Belief Desire Intention (BDI) \[64\] agent architecture model is a powerful high-level decomposition and abstraction tool in analyzing, designing, and implementing complex software systems which has two important features - event-driven and means-ends reasoning. There exist many agent frameworks realizing BDI architecture such as JAM[44], PRS[33], ACT[56], SCS[49], AgentTCL[35], ARA[57], JADEX[59], JACK[43] etc.

These frameworks provide mature agent-oriented software development process for multiagent systems. The BDI model concentrates on the roles of the intentions in practical reasoning. Practical reasoning is reasoning directed towards actions. Practical reasoning consists of two activities:

- Deliberation (deciding what state of affairs we want to achieve) and the outputs of deliberation are intentions
- Means-ends reasoning (deciding how to achieve these states of affairs) and the outputs of means-ends reasoning are plans.

Most BDI platforms share the following three features.

- An agent contains four key data structures - Beliefs, Goals, Plans and Intentions. Beliefs are the informational state representing what an agent knows about itself and the world which may be incomplete or even incorrect. Thus beliefs represent the agents current knowledge about the world, including information about the current state of the environment inferred from perception devices and messages from other agents, as well as internal information. Goals or desires are the motivational state and correspond to what the agent wants to achieve. Plans
represent the procedural knowledge about how to achieve a certain goal or react to a specific situation. Thus a plan library is a set of *recipes* representing the procedural knowledge of the agent. *Intentions* are selected plans for execution and represent the deliberative state of an agent. Thus intentions are the chosen means to achieve the agent's desires, and are generally implemented as plans and post-conditions.

- The execution of an agent is event driven. There is an event queue where both events (either perceived from the environment or generated by the agent itself to notify an update of its belief base) and internal subgoals (generated by the agent itself while trying to achieve a desire) are stored. Plans, usually denoted as event preconditions — action sequence, are defined to react to a certain event which can be internal modifications to its goals and beliefs or external changes of environment. After the event is triggered, the preconditions will be tested before action sequence can be chosen to execute. Because events can occur non-deterministically, plans are executed reactively.

- The execution path to achieve a goal of an agent is generated by means-end reasoning. That is, the goal is treated as an initial event triggering a corresponding plan to run. action sequence of the plan may contain primitive actions as well as sub-goals. All sub-goals will be in turn treated as events to trigger sub-plans to run. This process continues recursively until all actions in sub-plans are primitive or atomic.

BDI agent architecture contains two major processes: *deliberation*, which helps deciding what goals the agent should achieve; *means-ends*, which finds the actual ways of achieving these goals. Before the execution of any actions, a BDI agent should determine goals and the available options. Once some options are chosen, they are
committed and the chosen options become intentions, inducing the agents actions. Intentions might also be fed back into the agents future practical reasoning. As in general, an agent may have multiple desires, an agent can have a number of intentions active at any one time. These intentions may be thought of as running concurrently, with one chosen intention active at any one time. In the multi-agent systems (MAS) community, each agent is given the mandate to achieve defined goals. To do this, it autonomously selects appropriate actions, depending on the prevailing conditions in the environment, based on its own capabilities and means until it succeeds, fails, needs decisions or new instructions or is stopped by its owner.

The core of a BDI agent is the interpreter. The interpreter will update the beliefs based on acquired information retrieved from the environment or the information passed from other agents. Desires will be chosen from the desire set based on the beliefs. An adequate plan is chosen based on the beliefs. The plan is executed in order to accomplish the designed intentions. The interaction between agents will be carried on continuously until the global goal is reached.

The typical BDI execution cycle is characterised by the following steps[63]:

1. Observe the world and the agents internal state, and update the event queue to reflect the events that have been observed;
2. generate new possible desires (tasks), by finding plans whose trigger event matches an event in the event queue;
3. from this set of matching plans, select one for execution (an intended means);
4. push the intended means onto an existing or new intention stack, according to whether or not the event is a subgoal; and
2.1. Agents

5. select an intention stack, take the topmost plan (the intended means), and execute the next step of this current plan. If the step is an action, perform it, otherwise, if it is a subgoal, post this subgoal on the event queue.

Usually, BDI-style agents do no first principles planning at all, as all plans must be generated by the agent programmer at design time. The planning done by agents consists entirely of context-sensitive subgoal expansion, which is deferred until a point in time at which the subgoal is selected for execution.

2.1.3 BDI-Based Languages and Platforms

In this section an overview of various BDI-style languages and MAS platforms is being provided. Note that several of these languages are being updated and newer features are being added regularly.

2.1.3.1 BDI languages

The Procedural Reasoning System (PRS) [33] is one of the first implementation in lisp language based on BDI architecture developed by SRI International. The system is developed as a representation of an experts procedural reasoning. It is used for evaluating maintenance procedures for the space shuttle in a simulation. After the success of PRS, it is rewritten in C++ known as distributed Multi-Agent Reasoning System (dMARS) [24] at Australian AI Institute. The implementation of the platform also includes graphical editors, compiler and interpreter for a goal-oriented logical language. dMARS has been successfully used in industrial applications and agent-based simulation systems.

AgentSpeak(L) [61] starts by evolving from PRS and dMARS and formalizes its operation. It is based on a restricted first-order language with events and actions.
2.1. Agents

Jason \cite{9} is one of the implementations of AgentSpeak(L) that extends with a variety of features. AgentSpeak(L) and its interpreter Jason has been chosen as the base language for implementing the concepts described in this thesis. A detailed overview of AgentSpeak(L) is given in the next section.

3APL \cite{38} architecture has many similarities with other architectures such as PRS. Different from other architectures, 3APL is designed to control and revise todo goals of agent. 3APL also incorporates practical reasoning rules to revise mental attitudes. A 3APL agent is defined with a set of actions and a set of rules. 3APL supports the design and construction of intelligent agents for the development of complex systems through a set of intuitive concepts like beliefs, goals and plans. The deliberation cycle in 3APL is a programmable component.

GOAL \cite{39} is an agent programming language for programming rational agents. The language provides the basic building blocks to design and implement rational agents by means of a set of programming constructs. The language provides an intuitive programming framework based on common sense or practical reasoning. The distinguishing feature of GOAL are the concept of a declarative goal. Goals of a GOAL agent describe what an agent wants to achieve, not how to achieve it. Different from other languages, GOAL agents are committed to their goals and only remove a goal when it has been completely achieved.

Dribble \cite{79} is a propositional language that constitutes a synthesis between the declarative features of the language GOAL, and the procedural features of 3APL. The important feature of Dribble compared with the original version of 3APL, is the addition of declarative goals which is based on GOAL.

Coo-BDI (Cooperative BDI) \cite{3} is based on the dMARS specification and extends it by introducing cooperations among agents to retrieve external plans for achieving
2.1. Agents
desires. The cooperation strategy is defined by a set of agents to cooperate, plan
retrieval policy and plan acquisition policy. The mechanism for retrieving relevant
external plans involves cooperation with trusted agents.

CAN [81] is a high-level agent language, in a spirit similar to that of AgentSpeak
[61]. CAN provides an explicit goal construct that captures both the declarative and
procedural aspects of a goal. Goals are persistent in CAN in that, when a plan fails,
another applicable plan is attempted.

2.1.3.2 BDI platforms

JAM [44] is an intelligent agent platform that combines ideas drawn from the BDI
theories and several other systems including PRS [33], SRI Internationals ACT plan
interlingua [56], and Structured Circuit Semantics (SCS) representation [49]. It also
addresses mobility aspects from Agent Tcl [35], Agents for Remote Action (ARA) [57]
and others.

JACK [43] is a commercial and mature Java implementation of BDI architecture.
Jack introduces agent oriented programming concepts on top of object oriented Java
language and supplies a runtime to support this agent oriented extensions. Agent
definitions can use notion of capability that allows modularity by encapsulation and
promotes code/design reuse. Jack treats goals as a special kind of event in its event
based interpretation.

Jadex [59] is an open source BDI architecture implementation in Java. Jadex
emphasizes use of goal as a first class data structure unlike other BDI implementation
where a goal is treated as a special event type whose handling results in plan activation.
Jadex is a BDI interpreter that is not bound to underlying agent software middleware.
Jadex is also flexible in terms of runtime adaptability which allows an agent to add
any belief, goal and plan definition in runtime.
2.1.4 AgentSpeak(L)

AgentSpeak(L) is a agent language with explicit representations of beliefs and intentions for agents that was introduced by Rao [61]. AgentSpeak(L) is a programming language based on a restricted first-order language with events and actions. The behaviour of the agent (i.e., its interaction with the environment) is dictated by the programs written in AgentSpeak(L). The beliefs, desires, and intentions of the agent are ascribed to agents written in AgentSpeak(L). The language construct The current state of the agent, which is a model of itself, its environment, and other agents, can be viewed as its current belief state; states which the agent wants to bring about based on its external or internal stimuli can be viewed as desires; and the adoption of programs to satisfy such stimuli can be viewed as intentions.

The basic operation of agents in AgentSpeak(L) is based around their beliefs, desires and intentions. An agent has beliefs (about itself, others and the environment), desires (in terms of the states it wants to achieve in response) and intentions as adopted plans. A goal is a state of the system which the agent wants to bring about. There are two types of goals in AgentSpeak(L). An achievement goal states that the agent wishes to achieve a state of the world in which the associated predicate is true. A test goal, states that the agent wishes to test if the associated predicate is a true. When an agent acquires a new goal or notices a change in its environment, it may trigger additions or deletions to its goals or beliefs. These events are called triggering events. Addition (denoted by +) and deletion (denoted by -) of beliefs and goals are the four triggering events. The purpose of an agent is to observe the environment, and based on its observation and its goals, execute certain actions. These actions may change the state of the environment.

In addition, agents also maintain a repository of available plans, known as the plan library. Plans are the central concept to the abilities of an agent. They are means that
enable an agent to respond to the changes in its’ environment. An agent has plans which specify the means by which an agent should satisfy an end. A plan consists of a head and a body. The head of a plan consists of a triggering event and a context, separated by a “:”. The triggering event specifies why the plan was triggered, i.e., the addition or deletion of a belief or goal. The context of a plan specifies those beliefs that should hold in the agent’s set of base beliefs, when the plan is triggered. The body of a plan is a sequence of goals or actions. It specifies the goals the agent should achieve or test, and the actions the agent should execute. It is to be noted that events in AgentSpeak(L) might be external or internal. External events represent the changes in the state of the world that should be handled by the agent. On the other hand, internal events are triggered from within the agent as a result of executing a plan. Agents respond to changes in their goals and beliefs, which result from perception, and which are packaged into data structures called events, representing either new beliefs or new goals. They respond to these changes by selecting plans from the plan repository for each change and then instantiating one of these plans as an intention. These intentions comprise actions and goals or plans to be achieved, with the latter possibly giving rise to the addition of new plans to that intention.

An AgentSpeak(L) agent is described as a tuple \( \langle E, B, P, I, A, S_E, S_O, S_I \rangle \) where \( E \) is a set of events, \( B \) is a set of base beliefs, \( P \) is a set of plans, \( I \) is a set of intentions, \( A \) is a set of atomic actions, \( S_E \) selects an event from the set \( E \), \( S_O \) selects a plan from the set \( P \) and \( S_I \) selects an intention from the set \( I \).

The alphabet of the formal language consists of variables, constants, function symbols, predicate symbols, action symbols, connectives, quantifiers, and punctuation symbols. Apart from firstorder connectives, AgentSpeak(L) uses ! (for achievement), ? (for test), ; (for sequencing), and ← (for implication).

An AgentSpeak(L) program is a collection of clauses of the form:
event : belief₁ & belief₂ & ... & beliefₙ ← actions/subgoals

As an example, consider a plan associated with the triggering event !move(O,A,B) corresponding to an achievement goal to move an object O from A to B, where:

- e is !move(O,A,B);
- at(O,A) and not at(O,B) are belief literals; and
- -at(O,A) and +at(O,B) are two steps in the plan body, consisting of information about belief additions and deletions.

The plan then is as follows:

+!move(O,A,B) : at(O,A)&notat(O,B) ← -at(O,A); +at(O,B).

When this plan is executed, it should result in the agent believing O is no longer in position A, and then believing it is in position B. For an agent to rationally want to move O from A to B, it must believe O is at position A and not already at position B.

A typical AgentSpeak(L) execution cycle is characterized by the following steps.

1. observe the world and the agent’s internal state, and update the event queue consequently; $S_E$ selects an event from the set $E$;

2. generate possible new plan instances whose trigger event matches an event in the event queue (relevant plan instances) and whose precondition (beliefs) is satisfied (applicable plan instances); plan selection ($S_O$) is based on the satisfiability of the current set of beliefs; if there are several plans, one is chosen nondeterministically;

3. select for execution one instance from the set of applicable plan instances;
4. push the selected instance onto an existing or new intention stack, according to whether or not the event is a (sub)goal;

5. select an intention stack using the selection function $S_I$, take the topmost plan instance and execute the next step of this current instance: if the step is an action, perform it, otherwise, if it is a subgoal, insert it on the event queue.

2.1.4.1 Jason: An interpreter of AgentSpeak(L)

One of the most popular fully-fledged interpreter of AgentSpeak(L) is Jason [9]. Jason has many extensions making up for a very expressive programming language for cognitive agents. It implements the operational semantics of that language, and provides a platform for the development of multi-agent systems, with many user-customizable features. Jason is implemented in Java (thus multi-platform) and is available Open Source, distributed under GNU LGPL.

Jason allows inter-agent communication and supports plan failure handling through the use of negative triggering events. Jason also features external environments, programmable in Java. Event and message handling are implemented using the event-driven plan rules. Plan failure handling is implemented similarly: The failure of a plan is represented by the removal of its goal, which is a negative triggering event. Plan rules, triggered by these negative triggering events, are used to implement plan failure. Jason also supports multi-agent system functionality. A separate .mas2j file can be created, describing the names, architectures and amounts of the agents to start and any external environments to be used. Also, Jason plan rules can feature a plan label. This label is noted after the plan rule’s Head and can take the form of a predicate, helping the interpreter choose one plan over another. For example, it can contain information about the plan, such as its cost or payoff. Also, when an agent receives a
message, the name of the sender is included as a plan label. This label can then be used by the agent to send a reply.

The Jason interpreter is also customisable in many ways. Developers can specify custom beliefbases, plan selection functions or agent architectures, allowing agents to be adapted to various problem domains. By default, Jason features internal actions for sending messages, starting and stopping agents, adding or dropping goals and plans, checking whether a variable has been ground, generating random numbers and even retrieving the current time and date, among many other things. This library of internal actions is easily extendable, as new actions can be programmed in Java. A second type of action available for use in the body of plan is the addition of new events, which in turn create new (sub)goals and trigger new plans. For example, it allows customised agents to treat messages from certain agents with priority, only select a certain goal for execution when there is nothing else to be done or register the time a belief was added to the beliefbase as a plan label.

2.1.4.2 Option and Intention Selection

While most of the BDI languages and interpreters mention about option and intention selection, there has not been much work in literature on how to best chose a plan or an intention. The default option selection function simply returns the first option (plan or intention) in the list. In most of the cases, it is left to the programmer to define these functions in an arbitrary manner. In Jason [9], annotations in plan labels can be used for the implementation of sophisticated option selection functions. The functions can use expected utilities annotated in the plan labels to choose among alternative plans. Also, as the label is part of an instance of a plan in the set of intentions the annotations can be changed dynamically for the implementation of efficient intention selection functions. This approach is similar to that in AgentSpeak(XL) [8]. However,
these approaches rely heavily on the knowledge of the domain expert and no universal method is suggested. In another approach, [1], extensions to Jaosn are made where domain independent meta-level reasoning is used for plan and intention selection. Here a data repository called the tool base (TB) exists besides the Plan Library (PL). TB is populated pro-actively at the run time by special percepts unlike the PL which is defined at design time. Using the TB and PL, the agent chooses the best plan and intention based on the heuristics rules that are define here.

2.2 Structured Operational Semantics

Structural operational semantics (SOS) provides a framework for giving operational semantics to programming and specification languages. Because of its intuitive appeal and flexibility, a growing number of programming languages from commercial and academic spheres have been given usable semantic descriptions by means of structural operational semantics. Moreover, it is becoming a viable alternative to denotational semantics in the static analysis of programs, and in proving compiler correctness. This section gives an introduction to SOS which has been used to describe our agent programming language in later chapter.

SOS was introduced by Gordon Plotkin in [58] as a logical means to defining operational semantics. The basic idea behind SOS is to define the behavior of a program in terms of the behavior of its parts, thus providing a structural, i.e., syntax oriented and inductive, view on operational semantics. An SOS specification defines the behavior of a program in terms of a (set of) transition relation(s). SOS specifications take the form of a set of inference rules which define the valid transitions of a composite piece of syntax in terms of the transitions of its components.

SOS provides a framework to give an operational semantics to programming and
specification languages, which, because of its intuitive appeal, has found considerable
application in the theory of concurrent processes. In particular, SOS has been success-
fully applied as a formal tool to establish results that hold for whole classes of process
description languages.

SOS generates a labeled transition system, whose states are the closed terms over
an algebraic signature, and whose transitions between states are obtained inductively
from a collection of so-called transition rules of the form $\frac{\text{premise}}{\text{conclusion}}$. A transition
basically takes a system from a configuration to a subsequent configuration. The
operational semantics for a programming language describes how a valid program is
interpreted as sequences of computational steps. These sequences then are the meaning
of the program. In the context of functional programs, the final step in a terminating
sequence returns the value of the program.

A typical example of a transition rule is $\frac{x \sigma x'}{x \parallel y \sigma x' \parallel y}$ stipulating that if $t \sigma t'$ holds
for certain closed terms $t$ and $t'$, then so does $t \parallel u \sigma t' \parallel u$ for each closed term $u$.
In general, validity of the premises of a transition rule, under a certain substitution,
implies validity of the conclusion of this rule under the same substitution.

To look at another example, let $C_1, C_2$ range over programs of the language, and
let $s$ range over states (e.g. functions from memory locations to values). If we have
expressions (ranged over by $E$), values ($V$) and locations ($L$), then a memory update
command would have semantics:

$$\frac{}{\langle E, s \rangle \Rightarrow V}$$
$$\frac{\langle L := E, s \rangle \Rightarrow (s \cup (L \mapsto V))}{}$$

Informally, the rule says that if the expression $E$ in state $s$ reduces to value $V$, then
the program $L := E$ will update the state $s$ with the assignment $L = V$.

The semantics of sequencing can be given by the following three rules:
2.3 Constraint Programming

Informally, the first rule says that, if program $C_1$ in state $s$ finishes in state $s'$, then the program $C_1;C_2$ in state $s$ will reduce to the program $C_2$ in state $s'$. The second rule says that if the program $C_1$ in state $s$ can reduce to the program $C_1'$ with state $s'$, then the program $C_1;C_2$ in state $s$ will reduce to the program $C_1'C_2$ in state $s'$. The semantics is structural, because the meaning of the sequential program $C_1;C_2$, is defined by the meaning of $C_1$ and the meaning of $C_2$.

2.3 Constraint Programming

In this section I give a brief overview of constraint programming which is one the key focus area of the thesis. This background will lay the foundation of how agents could be used as optimisers.

2.3.1 Constraint Satisfaction and Optimisation Problem

Constraint Satisfaction Problems (CSPs) \[36], \[76] are a very powerful and general knowledge representation formalism, since many real situations can be described by a set of objects, together with some constraints among them. They were a subject of research in Artificial Intelligence for many years starting from the 1970s. Many CSPs require a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a vector of variables that satisfies all constraints.

A constraint satisfaction problem prescribes some requirements for a finite number of variables in the form of constraints. The set of possible values the domains for each
variable is finite. A constraint tells which value tuples are allowed for a certain subset of all the variables. A constraint can be given either explicitly, by enumerating the tuples allowed, or implicitly, e.g., by an algebraic expression.

Definition 1. A constraint satisfaction problem (CSP) is a triple $P = (V, D, C)$, where

- $V = \{v_1, \cdots, v_n\}$ is the set of variables called domain variables;
- $D = \{D_1, \cdots, D_n\}$ is the set of domains. Each domain is a finite set containing the possible values for the corresponding variable;
- $C = \{c_1, \cdots, c_n\}$ is the set of constraints. A constraint $c_i$ is a relation defined on a subset $\{v_{i1}, \cdots, v_{ik}\}$ of all the variables, that is $\{D_{i1}X \cdots XD_{ik}\} \supseteq c_i$.

Within the search for solution we may decide for different kinds of exploration of the solution space. We may search for one solutions, all solution, or often for some best solution with respect to given criteria. Such solution is searched within the so called constraint satisfaction optimization problem (CSOP). Thus a CSOP is a CSP with an objective function $F$ which usually evaluates to a real number and whose value should be minimized or maximized.

Problems ranging from map coloring, vision, robotics, job-shop scheduling, VLSI design, and so on, can easily be cast as CSPs and solved using one of the many techniques that have been developed for such problems or subclasses of them [30], [31], [55].

2.3.2 Constraint Logic Programming (CLP)

Constraint Logic Programming (CLP) as a language comes from the integration of consistency techniques in Logic Programming in the mid-eighties. In a pure logic
programming language such as the early PROLOG, the constraint store contains only equations that unify predicates. One obtains CLP by expanding the repertory of constraints and variables. Marriott and Stuckey \cite{54} identify the first CLP system to be PROLOG II \cite{14}, in which unification requires solution of disequations as well as equations. Jaffar and Lassez \cite{5} pointed out that PROLOG II is a special case of a general scheme in which unification is viewed as a constraint-solving problem. The term \textit{constraint logic programming} originates from this paper. There has been many CLP systems since its inception.

One of the central themes of logic programming is to combine the declarative and the procedural. A logic program can be read two ways: as a series of logical propositions that state conditions a solution must satisfy, and as instructions for how to search for a solution. Constraint logic programming (CLP) can be conceived generally as the embedding of constraints within a programming language. This combination of declarative and procedural modeling gives the user some control over how the problem is solved, even while retaining the ability to state constraints declaratively. The power of recursion allows one to define a wide range of constraints in a logic program, but solution can be inefficient, particularly where numerical operators are involved. To solve many practical problems, one must exploit the special properties of constraints, for instance, by using linear solvers or interval arithmetic for inequalities and specialized propagation schemes for all-different constraints and CLP addresses this problem. Its key idea is to regard the unification step of logic programming as a special case of constraint solving.

CLP aims at combining the declarative aspects of logic programming and constraint solving in an efficient problem solving environment. Constraints over different domains can be stated in a uniform framework and are solved with methods originating in various areas, from artificial intelligence (AI) to Operations Research (OR).
CLP is now used industrially for applications as diverse as digital circuit design, portfolio management and production scheduling systems and there are several industrial systems available for constraint programming.

CLP programs are of the form:

\[
\text{solve(Variables) :- initialize variables(Variables), state constraints(Variables), search(Variables) where initialize variables and state constrains define the model (declarative part of the solution) and search defines the control part of the solution.}
\]

In a timetabling problem, where classrooms and times are assigned to instructors, the CLP would consist of time and classroom variables, the constraints (hard) would be disjunctive\((\text{Time1, Time2})\) which would ensure different times for one instructor teaching two classes defined by \text{Time1} and \text{Time2} and search would find when (assign time variables) and where (assign classroom variables) classes must be taught.

The essence of Constraint Programming is based on a clean separation between the statement of the problem (the variables and the constraints), and the resolution of the problem (the algorithms). The basic way to express and model a combinatorial problem is with:

- Variables, which are defined with a specified domain and should allow to model the possible decisions one can take for determining a solution to the problem.

- Constraints, describing all the restrictions on variables and all relations between these variables that must be satisfied for the given problem. The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship.

- Objective function(s), specifying what must be optimized. An objective function is usually expressed as a function of (a part of) the decision variables or related
to the variables in a more implicit or complex manner.

2.3.3 CLP toolkits

There has been several commercially successful CLP systems that model and solve real-life problems. The first notable system among the CLP toolkits is CHIP [23]. Other CLP systems include ILOG [27], Prolog III [15], CLP(R) [46], ECLiPSe [4], CIAO [37] and clp(fd) [13] which have been and are being used in many commercial applications. In the next section I give an overview of Eclipse which has been used in the implementation of our agent programming language.

2.3.3.1 ECLiPSe

ECLiPSe [4] is a software system for the cost-effective development and deployment of constraint programming applications, e.g. in the areas of planning, scheduling, resource allocation, timetabling, transport etc. It is also ideal for teaching most aspects of combinatorial problem solving, e.g. problem modeling, constraint programming, mathematical programming, and search techniques. It contains several constraint solver libraries, a high-level modeling and control language, interfaces to third-party solvers, an integrated development environment and interfaces for embedding into host environments. ECLiPSe supports the independent development of new constraint solvers. This can also be done by building on the existing ones. Furthermore the (attributed) variable data type is the key to many extensions to the basic Prolog language. The system calls user-definable event handlers when it encounters attributed variables in certain contexts, like unification. ECLiPSe offers solving in finite domain as well continuous domains in the form of numeric intervals. One of the most important libraries that ECLiPSe offers are eplex - which allows the integration of Mathematical
Programming techniques with its native Constraint Logic Programming techniques within the same unified framework and IC (Interval Constraint) library - which is a hybrid integer/real interval arithmetic constraint solver. ic solves equations and inequations between general arithmetic expressions over continuous or integral variables and the expressions can include non-linear functions such as sin, built-in constants such as pi.

### 2.3.3.2 Example

One of the most important available packages available in ECLiPSe for solving linear and mixed integer problems which support optimisation is the eplex library. In the travelling salesman problem, the decisions of what order to visit the cities are based on optimising the total distance travelled by the salesman and can modeled as a transportation problem, which uses the eplex library to pass the constraints to the CPLEX package. Below I give an example of a design model of a transportation problem and is taken from [80].

```prolog
:- lib(eplex).

main(Cost, Vars) :-

    % Transportation problem: clients A,B,C,D, plants 1,2,3.
    % Variables represent amount delivered from plant to client.

    Vars = [A1, B1, C1, D1, A2, B2, C2, D2, A3, B3, C3, D3],
    Vars :: 0.0..10000.0,          % variables

    % ...
```


\[ A_1 + A_2 + A_3 = 200, \quad \% \text{client demand constraints} \]
\[ B_1 + B_2 + B_3 = 400, \]
\[ C_1 + C_2 + C_3 = 300, \]
\[ D_1 + D_2 + D_3 = 100, \]
\[ A_1 + B_1 + C_1 + D_1 \leq 500, \quad \% \text{plant capacity constraints} \]
\[ A_2 + B_2 + C_2 + D_2 \leq 300, \]
\[ A_3 + B_3 + C_3 + D_3 \leq 400, \]

\text{optimize}(\text{min}(\quad \% \text{solve minimizing})
\[ 10A_1 + 7A_2 + 11A_3 + \quad \% \text{transportation costs} \]
\[ 8B_1 + 5B_2 + 10B_3 + \]
\[ 5C_1 + 5C_2 + 8C_3 + \]
\[ 9D_1 + 3D_2 + 7D_3), \text{Cost}). \]

The answer returned by ECLiPSe is
\[ C = 6600.0 \quad V = [0.0, 200.0, 300.0, 0.0, 0.0, 200.0, 0.0, 100.0, 200.0, 0.0, 0.0, 0.0, 0.0] \]

2.3.4 Soft Constraints

Classical constraint satisfaction problems (CSPs) \cite{52} are a very expressive and natural formalism to specify many kinds of real-life problems. However, CSPs are not able to model constraints that are preferences rather than strict requirements or to provide a fairly okay solution when the problem is over-constrained \cite{6}.

Soft constraints model quantitative preferences by generalizing the traditional formalism of hard constraints. In a soft constraint, each assignment to the variables
of a constraint is annotated with a level of its desirability, and the desirability of a complete assignment is computed by a combination operator applied to the local preference values. By choosing a specific combination operator and an ordered set of levels of desirability, a specific class of soft constraints can be selected. A soft constraint is a constraint that rather than returning a Boolean yields more informative values such as a preference value or a cost. Soft Constraints can be described by the type of ordering or scoring that can be done. It can be classified in to two types basically:

- Assigning values to each possible tuple in a constraint
- Assigning a value to the actual constraint itself.

Soft Constraints can be modeled in a number of ways using these two methods.

- Fuzzy CSP’s \[65\], \[25\] allow constraint tuples to have an associated preference, like for example 1 = best and 0 = worst.
- Weighted CSPs (WCSPs)\[70\] tuples come with an associated cost. This allows one to model optimization problems where the goal is to minimize the total cost (time, space, number of resources etc.) of the proposed solution
- Probabilistic CSPs (Prob-CSPs) \[29\] model those situations where each constraint has a certain probability \( p(c) \), independent from the probability of the other constraints, to be part of the given problem.

2.3.4.1 Semiring based constraint satisfaction

In this section I discuss semiring based constraints which I have used to model user preferences in an agent.
A more general way of explaining over-constrained problems has been proposed by Bistarelli et.al [7] called as the Semiring-based Constraint Satisfaction problem. Here, a semi-ring (that is, a domain plus two operations satisfying certain properties) is all that is needed to satisfy to describe many constraint satisfaction schemes. In fact the domain of the semiring provides the levels of consistency which can be interpreted as cost or degrees of preferences or probabilities or others and the two operations define a way to combine constraints together. More precisely we define the notion of constraint solving over any semiring. Specific choices of semiring will then give rise to different instances of the framework which may correspond to new constraint solving schemes.

This framework uses a semiring structure, where the set of semiring specifies the preference associated to each tuple of values. The two semiring operations (⊕ and ⊗) then model constraint projection and combination respectively.

In semiring-based constraint satisfaction, each tuple in the constraint is marked by a preference level expressing how good the tuple satisfies the constraint. The preference level is taken from a set A equipped with the c-semiring structure \((A,⊕,⊗,0,1)\). A is a set of preferences, \(⊕\) is a commutative, associative, idempotent \((a ⊕ a = a)\) binary operation on A with the unit element 0 \((0 ⊕ a = a)\) and the absorbing element 1 \((1 ⊕ a = 1)\), \(⊗\) is a commutative, associative binary operation on A with the unit element 1 \((1 ⊗ a = a)\) and the absorbing element 0 \((0 ⊗ a = 0)\) and \(⊗\) distributes over \(⊕\).

The multiplication operation \(⊗\) is used to combine constraints. Let \(vars(c)\) be a set of variables over which the constraint \(c\) is defined, \(δc\) be a mapping of all tuples over \(vars(c)\) to A, i.e., \(δc(V)\) is a preference of the tuple \(V\) in the constraint \(c\), and let \(U ↓ Y\) be a projection of some tuple \(U\) to variables \(Y\). Then we can describe a preference of some tuple \(V\) by combining preferences of this tuple (its projection) in all the constraints \(C\):

\[
p(V) = \sum_{c ∈ C} δc(V ↓ vars(c))
\]
To compare the preferences of tuples we need some ordering on $A$. This ordering can be defined using the additive operation $\oplus$ in the following way: $a \leq b \iff a \oplus b = b$. If $a \leq b$ then we say that $b$ is better than $a$. Note that the relation $\leq$ defines a partial ordering on $A$ opposite to the total ordering used in the valued constraint satisfaction.

The semiring-based constraint satisfaction problem \cite{7} is defined formally by the $c$-semiring structure $(A, \oplus, \otimes, 0, 1)$, the set of variables $X$, their domains $D$, and the set of constraints $C$ described via $\delta_c$. The task is to find an assignment $V$ with the best preference $p(V)$. 
Chapter 3

Agent Programming with CASO

In this chapter I introduce a new agent language CASO which augments the reactive agent with rational thinking by applying decision making techniques into the execution cycle of an agent.

3.1 Motivating example

In order to describe our agent programming language called CASO, let us first consider an example which would help us to understand a detailed reasoning behind the adoption of CASO. Let us consider a simplified traveling salesman problem (TSP) where a salesman has appointments at 3 different places - A, B and C. The 6 possible orders of visiting them are A-B-C, A-C-B, B-A-C, B-C-A, C-B-A and C-A-B. In a typical TSP, the idea is to cover the places by traveling the short distance. The route can be planned via different planning algorithms which takes into account the distances. However, in real world, there are several constraints specifically in terms of time and cost. As an example, the salesman may need to reach A by 11 a.m., B by 2 p.m. and C by 3 p.m. There may be several transport options available - travel by taxi, train
or bus with each having different costs associated with them. Moreover, there are also fixed timings for buses and trains. Let us suppose that the main of the objectives of the salesman is to minimize transportation cost.

As we can see, this could be treated as an optimization problem the solution to which would be a possible path which the salesman would follow. But since things are never certain in the real world, there could be change of circumstances while the salesman is traveling. As an example, there may be road work on a particular road which would result in a bus taking an alternative route and cause delay. Again, appointment timings may be changed like visiting the place C may be postponed by 1 hr. Moreover, because of a competitor, the salesman may be told on the way that if he fails to reach B by 2 p.m. there may be a penalty of $10 for every hour delayed. These would then have to be factored into the objective function.

In the following sections, I outline the essential features of CASO and in context of our example.

### 3.2 Overview of CASO

CASO is a programming language based on AgentSpeak and is implemented using Jason. Its distinguishing feature happens to be the incorporation of constraints and objectives into the symbolic approach of BDI model.

Incorporating constraints into BDI languages can be of great advantage to agent decision making. Some of the advantages of using constraints are:

- Constraints have a well-defined intuitive semantics.
- Using constraints may lead to better expressive capabilities as well as more efficient computation (in some instances).
• Constraints can capture qualitative and quantitative preferences and costs.

• Constraints offer a declarative representation that is easy to understand.

• Constraints are supported by a large set of algorithms, solvers, and tools.

CASO incorporates Constraint Satisfaction and Optimization (CSOP) techniques where the optimization is based on the objective function (softgoal). This technique applies constraint and objective directed solving on the context section of a BDI agent’s plan specification in order to determine an applicable plan to fire. More interestingly, the use of constraint-based representations can make it possible to deal with explicit agent objectives (as distinct from agent goals) that express the things that an agent may seek to optimize at any given point in time. In CASO, one can express agent’s goals quantitatively - for example, agents can have some utility (objective) function which needs to be maximized. CASO also incorporates efficient option selection and intention selection (selecting the best plan to use to deal with the current event) with parametric look-ahead techniques, i.e., techniques where the extent of look-ahead style deliberation can be adjusted.

**Definition 2.** A CASO agent is a tuple: \( \langle \beta, P, E, I, \Theta, A, S_O, n, S_E, S_I, OS \rangle \)

where \( \beta \) is a set of beliefs which includes a set of constraints, \( P \) is a plan library, \( E \) is a set of events, \( I \) is a set of intentions, \( \Theta \) a set of objective functions, \( A \) is a set of basic actions with parameters, \( n \) is a parameter which determines how many steps the agent is going to look ahead before committing to a plan or intention, \( S_E \) is a selection function which selects an event to process from set \( E \) of events, \( S_O \) is a selection function which selects an applicable plan to a trigger \( t \) from set \( P \) of plans, \( S_I \) is a selection function which selects an intention to execute from set \( I \) of intentions and \( OS \) is an objective store which stores objective functions.
3.2. Overview of CASO

In the following sections I explain each of the terms mentioned in the above definition.

3.2.1 Belief Base ($\beta$)

$\beta$ consists of simple belief facts as well as Constraint Logic Programs (CLPs)\cite{45}. As we will see later, such an approach which combines the flexibility of logic with the power of search to provide high-level constructs for solving computationally hard problems can help an agent to choose a plan or intention intelligently. Note that CLPs are also helpful in reducing the number of beliefs explicitly stated.

In our example, some of the beliefs would be:

\[
\text{bus(loc}\_A,\text{loc}\_B, 3).  \\
\text{location(loc}\_B).  \\
\text{dist(A,5).}  \\
\text{reachingTime(loc}\_B, 2).  \\
\text{taxi(Fare, Distance).}
\]

In CLP this would be written as

\[
taxi(Fare, Distance):- Fare = 2.5 \times Distance + 2
\]

3.2.2 Set of plans ($P$)

$P$ is a repository which contains all the available pre-compiled plans for the agent to use. When a triggering event occurs, all the plans triggered by this event that can be executed in the current circumstances are retrieved. Below I define a CASO plan.

**Definition 3.** A CASO plan $p$ is of the form $t : b_1 \land b_2 \land \cdots \land b_n \land c_1 \land c_2 \land \cdots \land c_m \leftarrow sg_1, sg_2, \cdots, sg_k$ where $t$ is the trigger; each $b_i$ refers to a belief; each $c_i$ is an atomic constraint; each $s_g$ is either an atomic action or a subgoal.
For brevity I will use $BContext(p)$ to denote the belief context of plan $p$. Thus $BContext(p) \equiv b_1 \land b_2 \land \cdots \land b_n$. Similarly, I will use $CContext(p)$ to denote the constraint context of plan $p$. Thus $CContext(p) \equiv c_1 \land c_2 \land \cdots \land c_m$. It should be noted that in the definition of the plan above, an action could have parameters whose values are instantiated when the agent actually executes the plan.

In our example, the following three plans describe how he would reach X depending on his location as well as availability of bus or train. Note that $Fare$ is the parameter of the action $pay$ which is the amount that he has to pay to reach X.

\begin{verbatim}
 p1: +!reach(X) : location(Y) & bus(X,Y,Fare) & (X!=Y)
     <- pay(Fare); !board_bus()

 p2: +!reach(X) : location(Y) & taxi(Fare,Distance) & (X!=Y)
     & dist(X,Distance) <- pay(Fare); !board_taxi()

 p3: +!reach(X) : location(Y) & train(X,Y,Fare) & (X!=Y)
     <- pay(Fare); !board_train()
\end{verbatim}

3.2.3 Set of Objective Functions ($\Theta$)

$\Theta$ represent objective functions like $\text{maximize}(exp)$ or $\text{minimize}(exp)$ where $exp$ consists of global variables that are valid throughout the lifetime of the agent. Objectives represent quantitative measure of goals that the agent would like to achieve.

The objective function initially would be given by the following:

$\text{minimize}(Fare)$
3.2.4 Set of events ($E$)

$E$ is the set of events which could be external or internal. Agents talk to the external environment through events. The different types of external events which originate from perception of the agent’s environment:

1. Addition and deletion of beliefs (with constraints).
2. Addition and deletion of achievement goals.
3. Addition and deletion of test goals.
4. Addition and deletion of objectives.

The first three types of events are triggering events (where the context of the plan is matched with relevant plans), while the last one is non-triggering. Internal events are generated from the agent’s own execution of a plan (i.e., as a subgoal in a plan generates an event of the type addition of an achievement goal). An internal event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed on top of that intention).

Example of events are:

+reach(loc_A) // new external event to reach A
+minimize(late_hours*10) // addition of new objective
?location(loc_A) // test if currently at location A
+!location(loc_B) // belief that currently at B. -minimize(minimize(Fare)) // deletion of objective
3.2.5 Objective Store (OS)

OS a consistent set of objective functions and is updated in case a new objective comes in as an event. Below I give the formal definition of what it means by augmenting the OS.

**Definition 4.** Given an objective store OS and a new objective $f$, the result of augmenting OS with $f$, denoted by $OS^*_f$, is defined as $\gamma(\text{MaxCons}(OS \cup f))$ where $\gamma$ is a choice function and $\text{MaxCons}(X)$ is the set of all $x \subseteq X$ such that $x$ is consistent and there exists no $x'$ such that $x \subset x' \subseteq X$ and $x'$ is consistent.

The new OS is now given by $\gamma(\text{MaxCons}(OS \cup \overline{O}) \cap OS)$ where $\gamma$ is the choice function, and $\overline{O}$ is the negation of the objective $O$.

Formally a *consistent objective store* is defined as below.

**Definition 5.** Objectives $O_1$ and $O_2$ are inconsistent if and only if there exists a pair of solutions $S_1$ and $S_2$ such that $S_1$ is preferred over $S_2$ by $O_1$ and the reverse holds under $O_2$.

3.2.5.1 Objective Store Consistency

Many real-world problems involve decisions based on multiple and conflicting criteria. When the objective functions conflict with each other, no single solution can simultaneously optimize all objective function. The more objectives are involved the more complex is the optimization problem and the choice for the decision maker. Therefore not all objectives can be used and some of the objectives may have to be omitted. Moreover, multiple objective consistency is contingent on the current set of constraints. Because of various constraints, objectives may not be consistent. The consistency checker of the objective store has to detect the conflicting and non-conflicting
optimization criteria and remove the ones that are not be used. I describe below a simple mechanism by which objective consistency could be checked by using patterns of inconsistencies. There are several cases to consider in order to determine consistency of the objective functions as outlined below. Note that the objective function \( \max(f(x)) \) is equivalent to \( \min(-f(x)) \).

- Let \( OS \) currently consist of the objective function \( \min(f(x)) \) and the new objective function is \( \min(f(y)) \) where variables \( x \) and \( y \) are different, then both the objectives functions would remain in \( OS \) and the new objective function would be \( \min(f(x)) + \min(f(y)) \).

- Let \( OS \) currently consist of the objective function \( \min(f(x)) \) and the new objective function is \( \min(g(x)) \) where the same variable \( x \) is present in both the objective functions. If \( f(x) \) is of the form \( \sum_{i=1}^{m} a_i x^k \) and \( g(x) \) is of the form \( \sum_{i=1}^{n} b_i x^l \) where \( m \) and \( n \) are integers, then
  1. If all of \( a_i \) and \( b_i \) are positive real number and all of \( k \) and \( l \) are positive integers, then the equivalent objective function is \( \min(f(x)) + \min(g(x)) \) and both functions would remain in \( OS \)
  2. If any of \( a_i \) and \( b_i \) is a negative real numbers and/or any of \( k \) and \( l \) is a negative integer, then there is a conflict of objective functions in the \( OS \) and the algorithm would discard the old objective function and put the new one in \( OS \).

- Let \( OS \) currently consist of the objective function \( \min(f(x,y)) \) and the new objective function is \( \min(g(x,y)) \) where there are two variables \( x \) and \( y \) in both. For simplicity we consider two variables here but the method could easily be extended for any number of variables. If \( f(x,y) \) is of the form \( \sum_{i=1}^{m} a_i x^k \cdot y^l \) and \( g(x,y) \) is of the form \( \sum_{i=1}^{n} b_i x^k \cdot y^l \) where \( m \) and \( n \) are integers then
1. If all of \( a_i \) and \( b_i \) are positive real number and all of \( k \) and \( l \) are positive integers, then the equivalent objective function is \( \min(f(x, y)) + \min(g(x, y)) \) and both functions would remain in \( OS \).

2. If any of \( a_i \) and \( b_i \) is a negative real numbers and/or any of \( k \) and \( l \) is a negative integer, then there is a conflict of objective functions in the \( OS \) and the algorithm would discard the old objective function and put the new one in \( OS \).

In our example, the objective store initially contains \( \text{minimize}(\text{Fare}) \). After some time it would contain the following consistent objectives: \( \text{minimize}(\text{late hours} \times 10) \) and \( \text{minimize}(\text{Fare}) \). Suppose now we have a new objective of the type \( \text{maximize}(\text{Fare}) \) that comes in the event queue. This objective is in conflict with the objective \( \text{minimize}(\text{Fare}) \) in the \( OS \). This would result in an inconsistency in the \( OS \). The consistency checker would now discard the old objective and keep the new objective. Thus the \( OS \) would now have \( \text{minimize}(\text{late hours} \times 10) \) and \( \text{maximize}(\text{Fare}) \).

3.2.6 Set of intentions (\( I \))

Intentions are particular courses of actions to which an agent has committed in order to handle certain events. \( I \) consists of a set of intentions where each intention is a stack of partially instantiated plans.

3.2.7 Look ahead parameter (\( n \))

\( n \) is a parameter which determines how many steps the agent is going to look ahead before committing to a plan or intention. This is used in the option and intention selection functions which is described later.
3.2.8 Event Selection Function \((S_E)\)

\(S_E\) selects an event and updates \(OS\) in case it is an objective function and in case it is a triggering event it passes it on to the interpreter which would unify it with the set of triggering events in the heads of plans.

3.2.9 Option Selection Function \((S_O)\)

In CASO, goals are achieved by executing plans and each goal has at least one plan that can be used to satisfy the goal. Each plan can include sub-goals, but need not have any. The leaf nodes of the tree are plan-nodes with no children (i.e., no sub-goals). \(S_O\) selects a plan from \(P\) based on the current plan context, \(\beta\), \(OS\) and \(n\). The selection function is one of the most fundamental aspects of CASO and in the following subsections I explain it in detail.

3.2.9.1 Goal Plan Tree

We follow the method of using goal-plan tree given in [74] to decompose the set of plans into a tree structure where goals and plans are represented as nodes. A goal-plan tree is a bipartite directed graph, connecting (sub)goals with plans, and plans with subgoals. A goal-plan tree represents how the agent achieves the goal in which the children of each goal are alternative ways of achieving that goal (OR) whereas the children of each plan are sub-goals that must all be achieved in order for the plan to succeed (AND). The root of the tree is a goal. The children of a goal node are plans, representing the alternative plans that can be used for achieving the goal. The children of a plan node are goals, representing the subgoals of the plan. The root thus forms an agents top-level goal, while the other goal nodes represent subgoals of the
plans that can be used to achieve the top-level goal, or subgoals. For a plan to succeed all the subgoals and actions of the plan must be successful (AND); for a subgoal to succeed one of the plans to achieve it must succeed (OR).

### 3.2.9.2 Parametric look-ahead technique

Let us consider we have two applicable plans - P1 and P2. In order to determine which plan to choose the agent generates the goal-plan tree for all possible paths. The parameter $n$ creates the pseudo-leaf nodes and therefore we get distinct paths from root to these pseudo-leaf nodes. Figure 3.1 shows all the possible paths from root to pseudo-leaf nodes for the set of plans P1 to P10. The value of $n$ is 2 which means the goal-plan tree is expanded up to 2 levels.

Optimisation techniques are now applied by the optimizer to each of the applicable plan to determine an optimal solution. In effect we are solving a CSOP which consists of a standard CSP and an optimisation function that maps every solution (complete labelling of variables) to a numerical value. $S_O$ now chooses this optimal solution from that set. One of the properties of CASO is that since CSOP is solved at various steps using a solver, all the beliefs and constraints must be global variables. The CLP uses CSOP techniques to find the value of the objective function $O$ for each pseudo leaf. Next we choose the path which has the maximum value at the pseudo leaf based on the strategy chosen.

Plan selection is defined as follows:

**Definition 6.** Given a trigger $t$ and a set of applicable plans $AppPlans(t)$ for $t$, a plan $p \in AppPlans(t)$ is referred to as an $O$-preferred plan if and only if: $p \leq_{opt} p_i$ for all $p_i \in AppPlans(t)$.
3.2. Overview of CASO

Plan1: +!t : Context\(_1\) ← SG\(_1\); SG\(_2\).
Plan2: +!t : Context\(_2\) ← SG\(_3\); SG\(_4\).
Plan3: +SG\(_1\) : Context\(_3\) ← a\(_1\).
Plan4: +SG\(_1\) : Context\(_4\) ← a\(_2\).
Plan5: +SG\(_2\) : Context\(_5\) ← a\(_3\).
Plan6: +SG\(_2\) : Context\(_6\) ← a\(_4\).
Plan7: +SG\(_3\) : Context\(_7\) ← a\(_5\).
Plan8: +SG\(_3\) : Context\(_8\) ← a\(_6\).
Plan9: +SG\(_4\) : Context\(_9\) ← a\(_7\).
Plan10: +SG\(_4\) : Context\(_{10}\) ← a\(_8\).

- - - refers to AND nodes;
—– refers to OR nodes;
Context\(_i\) is the the context of Plan\(_i\) which is the conjunction of non-constraint and constraint predicates in ;
SG\(_i\) is subgoal for Plan\(_i\);
a\(_i\) is an atomic action for Plan\(_i\);

<table>
<thead>
<tr>
<th>Path id</th>
<th>Possible Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plan1-Plan3-Plan5</td>
</tr>
<tr>
<td>2</td>
<td>Plan1-Plan4-Plan5</td>
</tr>
<tr>
<td>3</td>
<td>Plan1-Plan3-Plan6</td>
</tr>
<tr>
<td>4</td>
<td>Plan1-Plan4-Plan6</td>
</tr>
<tr>
<td>5</td>
<td>Plan2-Plan7-Plan9</td>
</tr>
<tr>
<td>6</td>
<td>Plan2-Plan8-Plan9</td>
</tr>
<tr>
<td>7</td>
<td>Plan2-Plan7-Plan10</td>
</tr>
<tr>
<td>8</td>
<td>Plan2-Plan8-Plan10</td>
</tr>
</tbody>
</table>

Figure 3.1: Agent Plans and corresponding goal-plan tree
Selection of $O$-preferred plan can be further enhanced by using $n$ the look-ahead parameter form plan selection. In case $n=0$, no look-ahead is performed and maximizing the objective function on the set of applicable plans would result in an $O$-preferred plan as described earlier. However, if $n > 0$ then a look-ahead algorithm is performed to select the $O$-preferred plan.

Figure 3.2 describes the plan selection process with 1-step look-ahead to find the best possible plan from the set of possible plans as given in figure 3.1.

<table>
<thead>
<tr>
<th>For Plan1 we solve 4 CSOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CSOP#1 = \langle \beta + Context_1 + Context_3 + Context_5 + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#2 = \langle \beta + Context_1 + Context_3 + Context_6 + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#3 = \langle \beta + Context_1 + Context_4 + Context_5 + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#4 = \langle \beta + Context_1 + Context_4 + Context_6 + OS \rangle$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For Plan2 we solve 4 CSOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CSOP#5 = \langle \beta + Context_2 + Context_7 + Context_9 + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#6 = \langle \beta + Context_2 + Context_7 + Context_{10} + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#7 = \langle \beta + Context_2 + Context_8 + Context_9 + OS \rangle$</td>
</tr>
<tr>
<td>$CSOP#8 = \langle \beta + Context_2 + Context_8 + Context_{10} + OS \rangle$</td>
</tr>
</tbody>
</table>

Our selection of the best plan is given by the following where $\Theta(CSOP\#i)$ represents the optimal solution for $CSOP\#i$:

Compute $BestCase(Plan1) = \text{Max}(\Theta(CSOP\#1), \Theta(CSOP\#2), \Theta(CSOP\#3), \Theta(CSOP\#4))$ Compute $BestCase(Plan2) = \text{Max}(\Theta(CSOP\#5), \Theta(CSOP\#6), \Theta(CSOP\#7), \Theta(CSOP\#8))$

Choosing the best plan:
if $BestCase(Plan1)$ is greater than $BestCase(Plan2)$ choose Plan1 else choose Plan2

Figure 3.2: Plan Selection with 1 step look-ahead

In our TSP example, the plan which produces the least value of $Fare$ in $\text{minimize}(Fare)$ function would be chosen among plans p1, p2 and p3.

3.2.10 Intention Selection Function ($S_I$)

$S_I$ function selects one of the agent’s intentions, i.e., one of the independent stacks of partially instantiated plans within the set of intentions by applying techniques similar
to that of $S_O$. Each intention stack is a choice for the agent. In order to determine right intention, the agent first considers the top element (i.e., a partially instantiated plan) of every intention stack. Each of these intentions form a goal plan tree and here also we solve the CSOP which consists of $\beta$, $OS$ and $n$.

Like before, we have several choices and we can apply different strategy to choose a particular intention. However, one of the most notable difference with $S_O$ is that we instantiate the action parameters (if any) with values obtained from solving the CSOP.

### 3.3 Difference with Agentspeak(L)

Most of the syntax and semantics of CASO are similar to that of AgentSpeak(L) and its interpreter Jason. However, the most notable additions are:

1. A constraint directed technique is incorporated into the computation strategy employed during the interpretation process.

2. Plan context consists of conjunction of predicates some of which could be constraint predicates (unlike AgentSpeak(L)) which could be dealt with CLP machinery using specialized constraint solvers.

3. A look-ahead technique is now built into the system that helps the user to determine which particular plan or intention to select by setting the value of a look-ahead parameter.

4. An external event can be a triggering event as well as an addition or subtraction of an objective function.
3.4. Operational Semantics

5. A new data structures is added - an objective store to store the set of global objectives.

6. Unlike AgentSpeak(L), applicable plans are those relevant plans for which
   - there exists a substitution which, when composed with the relevant unifier and applied to the context, is a logical consequence of \( \beta \) and
   - the constraint predicates in the context of the plans are unified with \( \beta \) and dealt with the CLP machinery using specialized constraint solvers to determine if these are consistent.

7. The set of basic actions that the agent has to perform as part of an intention execution process may also contain parameters, the values of which may be set by the value of the constraint variables obtained from solving of a CSOP relevant to a given applicable plan. These values are instantiated during intention execution.

3.4 Operational Semantics

The operational semantics of CASO is defined using Plotkin’s Structural Operational Semantics [58]. We use a method similar to that shown in [51] which describes operational semantics of AgentSpeak(L). A CASO configuration \( C \) is a tuple \( C = \langle \beta, I, A, R, Ap, OS, i, \epsilon, \rho, \alpha \rangle \) where

- \( \beta \) is a set of beliefs
- \( I \) is a set of intentions \( \{i, i', \cdots\} \) and \( i \) is a stack of partially instantiated plans
- \( E \) is a set of events \( \{(te, i), (oe), (te', i'), (oe'), \cdots\} \) where
  1. a triggering event pair is denoted by \( (te, i) \) where \( te \) is the triggering event and the intention \( i \) has the plans associated with it;
(2) an objective event is denoted by oe which adds or removes element from the objective store;

- A is a set of actions \( \{(a_1, param_1'), \cdots, (a_2, param_2'), \cdots \} \). Each action is a tuple \( (a_i, param_i', \cdots) \) where \( a_i \) is the basic action and \( param_i', param_{i''}, \text{etc.} \) are the action parameters.

- R is a set of relevant plans.

- Ap is a set of applicable plans.

- OS is the objective store.

- Each configuration has 4 components denoted by \( \varsigma, \epsilon \) and \( \alpha \) which keep record of a particular intention, event and a set of action parameters associated with actions of an U-preferred plan respectively that are being considered along the execution of a plan.

In order to present the semantic rules for CASO I adopt the following notations:

- If \( C \) is a CASO configuration, I write \( C_E \) to make reference to the component \( E \) of \( C \). Similarly for other components of \( C \).

- I write \( C_{\varsigma} = \underline{\_} \) (the underline symbol) to indicate that there is no intention being considered in the agent’s execution. Similarly for \( C_\epsilon \) and \( C_\alpha \).

- I write \( i[p] \) to denote the intention that has plan \( p \) on its top.

- I write beliefs to denote the set of current beliefs together with the set of constraints.

The set of semantic rules related to Event, Plan and Intention selections are now given below.
3.4. Operational Semantics

**Event Selection:** $S_E$ selects events from a set of Events $E$. The selected event is removed from $E$ and is either assigned to the component $\epsilon$ of the configuration in case it is a triggering event or is added or removed to/from OS if it is an objective. The selected event is removed from $E$ and is assigned to the $\epsilon$ component of the configuration. Below I give the three semantic rules governing this function.

$$SelEv_1 \quad S_E(C_E) = (te, i) \quad C_\epsilon = \neg C_{AP} = C_R = \{\}$$
where: $C'_E = C_E - (te, i)$ and $C'_\epsilon = (te, i)$

$$SelEv_2 \quad S_E(C_R) = oe \quad C_\epsilon = \neg C_{AP} = C_R = \{\}$$
where: $C'_E = C_E - oe$, $C'_{OS} = C_{OS} + oe$ and $C'_\epsilon = oe$

**Option Selection:** $S_O$ selects a plan (p) from the set of applicable plans $Ap$. I also assume that there is a selection function $S_{OS}$ which selects a consistent objective store (ConsOS) where the maximal set of objectives that are consistent are kept and the rest are discarded.

$$SelOpr \quad S_O(C_{Ap}) = p \quad S_{OS}(C_{OS}) = ConsOS \quad C_\alpha = \neg C_\epsilon \neq \neg C_{Ap} = \{\}$$
where: $C'_{OS} = ConsOS$, $C'_\alpha \neq \{\}$ and $C'_{Ap} = \{\}$

**Creating Intentions:** Rule TrigEv says that if the event $\epsilon$ is external triggering event indicated by $T$ in the intention associated to $\epsilon$, a new intention is created and its single plan is the plan $p$. If the event is internal, rule IntEv says that the plan in $p$ should be put on top of the intention associated with the event. Either way, both the event and the plan can be discarded from the $\epsilon$ and $\iota$ components respectively.
3.5 Related Work

TrigEv \( \frac{C, \text{beliefs} \rightarrow C', \text{beliefs}}{C_e = (te, T)} \)
where: \( C'_i = C_i \cup \{[p]\} \), \( C'_e = _- \)

IntEv \( \frac{C, \text{beliefs} \rightarrow C', \text{beliefs}}{C_e = (te, T)} \)
where: \( C'_i = C_i \cup \{i[p]\} \), \( C'_e = _- \)

**Intention Selection:** \( S_I \) selects an intention for processing. The action parameters associated with the selected intention are initialized by solving the relevant CSOP in case there are hard constraints and objectives.

\( \frac{S_I(C_I) = i}{C_i = _-} \)
where: \( C'_i = i \)

In [61], Rao gives a one-to-one correspondence between the model theory, proof theory, and the abstract interpreter. CASO has been developed on the same principles of AgentSpeak(L) and thus similar proof theory can be applied for CASO and BDI framework.

### 3.5 Related Work

During the execution of an agent program, deliberations on various types of decision take place, like planning a task, selecting a goal if multiple goals are possible, deciding to execute it, and so on and are usually hard-coded into the programming language. The use of explicit constraints and objectives in BDI agent programming has not been looked into earlier. However, there has been quite a number of related research about goal planning, intention execution etc. which I briefly summarize below.
Most BDI agent programming languages like AgentSpeak(L) and 3APL do not perform lookahead or planning; actions are executed as soon as they are selected. Thus the BDI agent programming approach works well if good plans can be specified for all objectives that the agent may acquire and all contingencies that may arise. However, there are often too many possible objectives and contingencies for this to be practical. The agent can also search over the available plans by actually executing them, but this only works if choices can be undone. The look-ahead technique when applied to BDI agent may resemble decision tree where probability plays an important role. In decision analysis one analyses problems containing risk/uncertainty/probabilities. However, this work is different from decision analysis as there is no probability attached to each plan and because of uncertainties associated with agents beliefs the possibility of taking a particular course of action may change at any point of time. A practical application of decision analysis has been shown in [72] where partially observable Markov Decision Processes [11] (POMDPs) have been used in BDI model. POMDPs are Markov decision processes, with a set of states, a set of actions, a set of effects of these actions and a set of immediate values of these actions. In POMDPs, the actual state is not necessarily known to the agent. If utility functions are defined for sorting the practical reasoning rules, then uncertain beliefs can also be used as a probability distribution on the possible states in terms of POMDP, and the deliberation process can be modelled as a POMDP.

Dastini et.al. in [21] introduces a deliberation language based on 3APL [38] to give the agent designer full control over the deliberation process. The language also includes programming constructs to conditionally, respectively successively apply a meta-statement based on a belief formula. In [12] constraint logic programming and data model approach is used within BDI agent framework. However, this work speaks of BDI agents in general and does not integrate with any BDI programming lan-
3.5. Related Work

AgentSpeak(XL) programming language [8] integrates AgentSpeak (L) with the TAEMS scheduler in order to generate the intention selection function. It also describes a precise mechanism for allowing programmers to use events in order to handle plan failures which is not included in AgentSpeak(L). This work, however, adds priority to the tasks. Some related theoretical work on selecting new plans in the context of existing plans is presented in [42]. Another related work on detecting and resolving conflicts between plans in BDI agents is presented in [75]. The degree of boldness of an agent is defined in [71] which represents the maximum number of plan steps the agent executes before re-considering its intentions. However in this case it is assumed that the agent would backtrack if the environment changes after it has started executing the plans.
Chapter 4

Implementation details

The availability of constraint solvers and AgentSpeak(L) based agent programming languages provide features that are conducive to the development of CASO. This chapter gives an overview of the implementation details of CASO.

4.1 CASO Interpreter

The CASO interpreter is very similar to the Agentspeak(L) interpreter with main changes being related to the selection functions. The interpreter is clearly depicted in Figure 4.1. The interpreter manages a set of events, a set of beliefs, an objective store and a set of intentions with three selection functions.

The CASO interpreter matches the context of each plan with the external triggering event and this generates a set of all relevant plans. The context part of the plans is unified against the agents beliefs (which includes constraints). Constraint solving is now performed on these relevant plans to determine whether the constraint(s) in the context of the plan is (are) consistent with the belief base. This results in a set of applicable plans (plans that can actually be used at that moment for handling the
chosen event). $S_O$ now chooses the best applicable plan based on the look-ahead technique (described earlier) which becomes the intended means.

At every interpretation cycle of an agent program, CASO updates a list of events, which may be generated from perception of the environment, or from the execution of intentions (when subgoals are specified in the body of plans). It is assumed that beliefs are updated from perception and whenever there are changes in the agents beliefs, this
implies the insertion of an event in the set of events. On top of the selected intention there is a plan which would be chosen for execution. This implies that either a basic action is performed by the agent on its environment, an internal event is generated (in case the of an achievement goal denoted by $!g_i$), or a test goal is performed (which means that the set of beliefs has to be checked). If the intention is to perform a basic action or a test goal denoted by $?g_i$, the set of intentions needs to be updated. In the case of a test goal, the belief base will be searched for a belief atom that unifies with the predicate in the test goal. If that search succeeds, further variable instantiation will occur in the partially instantiated plan which contained that test goal (and the test goal itself is removed from the intention from which it was taken). In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention and initialize the parameters based on the solving the relevant CSOPs. When all formulae in the body of a plan have been removed (i.e., have been executed), the whole plan is removed from the intention, and so is the achievement goal that generated it (if that was the case). This ends a cycle of execution, and CASO starts all over again, checking the state of the environment after agents have acted upon it, generating the relevant events, and so forth.

4.1.1 Algorithm for CASO interpreter

An overview of the steps that the CASO interpreter follows in a single cycle is given below.

1. Perceiving the environment

2. Updating the belief bases

3. Select an Event from Event Queues
4.2 Integrating Jason

During thesis work, a proof of concept implementation of CASO is made. Since the architecture is based on AgentSpeak(L) [61], an implementation of its interpreter is needed that is suitable for extension and is open source. Jason [9], which is a popular and well known interpreter of AgentSpeak(L), is chosen because of these properties. Jason is a fully-fledged interpreter of AgentSpeak which includes speech-act based inter-agent communication. It is implemented in Java (thus multi-platform) and is available Open Source, distributed under GNU LGPL. The source code of Jason has been designed modularly, to be able to integrate it easily with multiple underlying agent platform architectures. The source code is stable and, although updates are still being released, all features are implemented.

Jason contains a lot of extensions to AgentSpeak(L) which helps the programmer to develop and deploy a MAS, some of which are not directly relevant for my extension purpose since I deal with a single agent. A summary of Jason’s extensions to the original AgentSpeak language which are pertinent to my work are given below:

- annotations in beliefs used for meta-level information and annotations in plan labels that can be used by selection functions;
• meta events, declarative goal annotations, higher order variables and treating
  plans as terms, imperative style commands in plan bodies, and various other
  language extensions;

• support for developing Environments;

• fully customisable (in Java) selection functions, trust functions, and overall agent
  architecture (perception, belief-revision, inter-agent communication, and acting);

### 4.2.1 Customizing Jason

In order to demonstrate the proof of concept of CASO, I modified Jason to incorporate
the features described in chapter 3. In CASO I concentrate on a single agent and hence
the features of Jason which deal with multiagents like communication etc. are not dealt
with. However, since I have kept the essence of Jason interpreter, the only notable
change I did has been in regards to the new operational semantics. In particular, my
main modifications to Jason are the following:

• CLP-style beliefs (with constraints and objectives) are now written in a separate
  file that is modified by an external event when a new objective is added or
  deleted. The ECLiPSe solver (described in detail in next section) reads this file
  and generates output.

• TransitionSystem.java which is part of the asSemantics package on Jason, is
  modified to call the external ECLiPSe solver and does the new file handling
  operation. It also implements the look-ahead function for option selection which
  can be added as a parameter.

• The syntax of a Jason program is changed to incorporate action parameters.
Thus actions in CASO contain parameters which are instantiated by the ECLiPSe solver and are used when the agent pursues intentions.

Thus any application that can be deployed in Jason can also be deployed in CASO with the added benefit of

- user-defined objective function that can change with every interpretation of CASO execution cycle by an external event;
- action parameters that are instantiated during execution of intentions.

4.3 Integrating ECLiPSe

I use ECLiPSe for option and intention selection function $S_O$. A CASO agent has a set of constraints/beliefs in its belief base and a set of objective functions at any point in time during the execution cycle. When an agent tries to select a plan from a possible set of applicable plans, it invokes the ECLiPSe constraint solver to determine the $O$-preferred plan. The ability for an user to add and remove objectives is a unique feature of a CASO agent which is not embedded inside the selection functions.

4.3.1 CASO execution cycle using CLP files

External ECLiPSe CLP files can be loaded into the CASO agent program by which the belief base of an agent is extended with the clauses defined in that file. These clauses are used to check the pre-conditions of each agent plan. Some of the agent beliefs could be written as CLP clauses while others are written as belief literals. The example in figure 4.2 shows an external ECLiPSe program which contains the predicate
resource_available() which form part of the belief base. Note that all variables defined by \textit{Vars} that are declared in the file must be global since these are shared across all CLP files.

\begin{verbatim}
resource_available() :-
Vars = [A1,A2,A3,B1,B2,B3,C1,C2,C3,D1,D2,D3],
Vars :: 0..inf,
integers(Vars),
(A1 + A2 + A3 = 21),
(B1 + B2 + B3 = 40),
(C1 + C2 + C3 = 34),
(D1 + D2 + D3 = 10),
(A1 + B1 + C1 + D1 =< 50),
(A2 + B2 + C2 + D2 =< 30),
(A3 + B3 + C3 + D3 =< 40).
\end{verbatim}

\textbf{Optimization function:}

\begin{verbatim}
optimize(max(1*A1 + 7*A2 + 200*A3 + 8*B1 + 5*B2 + 2*B3 + 5*C1 + 5*C2 + 1*C3 + 6*D1 + 4*D2 + 1*D3))
\end{verbatim}

Figure 4.2: CASO Belief and Objective function written in ECLiPSe CLP style

A problem in ECLiPSe is modeled by a set of simultaneous equations and could also contain an objective function that is to be minimized or maximized, subject to a set of constraints on the problem variables, expressed as equalities and inequalities. represent the set of variables. The ECLiPSe solver can be used for both linear constraints (\textit{eplex} library) and non-linear constraints (\textit{ic} library) or even a mix of both. Linear constraints are defined on the variables as shown below. The steps for executing ECLiPSe program is as follows:

- Generate possible new plan instances whose trigger event matches an event in the event queue (relevant plan instances) and whose precondition (beliefs) is satisfied (applicable plan instances). If the beliefs are simple literals then plan
selection ($S_O$) is based on the satisifiability of the current set of beliefs as in AgentSpeak(L). However if one of the preconditions correspond to a CLP, then external ECLiPSe CLP program is executed. In this example, if precondition is resource_available() then the program checks for the file resource_available.ecl.

- If the agent is trying to optimize an objective, then this objective function comes in the form of a non-triggering event. All the agent beliefs which are stored as CLP files are then updated with the new objective function.

- A simple consistency checking mechanism as described in chapter 3 is applied so that the objective does not conflict with existing objectives. If there is a conflict, then we discard old conflicting objectives and incorporate the new one; otherwise, the new object is added to the existing set of objectives.

- The result of executing the CLP along with the instantiated variables ($Vars$) and the value of the objective function (if any) is stored in an output file resource_available.txt.

- If the result of executing the CLP resource_available.ecl is false i.e., if the constraints are not satisfied, then the plan is not considered as an applicable plan. On the other hand, if the CLP returns a value true, then the plan becomes an applicable plan.

- The text file (resource_available.txt) containing the output from the ECLiPSe solver is read by $S_O$ and if there are several applicable plans to pursue, $S_O$ would choose the plan instance which produces the highest value of the objective function (in case of maximization) in for execution. In case there is no objective the $S_O$ picks up the first applicable plan.

- In case of look-ahead, the goal-plan tree is generated from the current node.
and all possible paths are explored by the look-ahead depth. This may require execution of several CLPs at the pseudo leaf level and choosing the best path based on the objective function.

- The selected instance is pushed onto an existing or new intention stack, according to whether or not the event is a (sub)goal.

- The selection function $S_I$ chooses one of the intention stacks to execute. This choice is based on picking up the topmost plan instance of each stack and executing the CLP corresponding to each plan precondition.

- As in case of $S_O$, the same logic is applied to select the stack. However, in this case, the result of executing the CLP along with the instantiated variables (Vars) and the value of the objective function (if any) is stored in an output file resource-available.txt.

- The topmost plan instance is used to execute the next step of this current instance: if the step is an action, perform it, otherwise, if it is a subgoal, insert it on the event queue. If the action contains parameters, then these are instantiated by the value of the variables from resource-available.txt.

4.4 Experimental Results

I ran a series of experiments to find out how the quickly the system could find the optimal plan. My goal is to show that with a reasonable number of look-ahead steps and with moderate number of plans/actions, the CASO agent would be reactive enough (i.e., perform plan selection in real-time). The experiments were conducted on Intel dual-core machine using complex set of constraints with linear objective functions which were basically solved by the ECLiPSe solver. The programs that were run were
generated randomly such that several goal-plan trees of various depths and heights
could be produced.

For any given CASO program, the following parameters can greatly affect the way
plan selection is done:

1. The branching factor of the goal-plan tree (i.e., the number of OR nodes that
   are present for each plan).

2. The look-ahead depth (or level) of the goal-plan tree up to which CSOP technique
   will be applied.

3. The number of constraints for each plan.

4. The number of variables in the CASO program.

5. The size of a CASO plan i.e., the number of subgoals it has (AND nodes).

I randomly generated CASO programs and tested the plan selection function by fixing
some parameters and varying other parameters as given above.

**Experiment 1:** Given a plan in a CASO program with multiple subgoals, I calcu-
lated the time taken to find the optimum plan among the choices by fixing the
branching factor, the number of subgoals, number of constraints per plan and
the number of variables for a given objective function and varying the depth of
look-ahead. I set branching factor = 3, the number of subgoals = 3, number of
constraints/plan = 2 and number of variables = 5.

**Experiment 2:** Given a plan in a CASO program with multiple subgoals, I calcu-
lated the time taken to find the optimum plan among the choices by fixing the
look-ahead depth, the number of subgoals, the branching factor, number of
4.4. Experimental Results

Figure 4.3: Graphs showing results of Experiment 1

constraints per plan variables and varying the number of variables. Note that for this experiment, I generated a number of CASO programs with the same set of plans having same head and body, but different context (different set of variables). Also, I used similar objective function with lesser variables. I set branching factor = 3, the number of subgoals=3, number of constraints/plan = 2 and look-ahead depth =3.

**Experiment 3:** Given a plan in a CASO program with multiple subgoals, I calculated the time taken to find the optimum plan among the choices by fixing the look-ahead depth, the number of subgoals, the branching factor, number of variables and the same objective function and varying the number of constraints per plan. Note that for this experiment, I generated a number of CASO programs with the same set of plans having same head and body, but different context (different number of constraints per plan). I set branching factor = 3, the number of
Figure 4.4: Graphs showing results of Experiment 2

subgoals=3, look-ahead depth = 3 and number of variables =5.

Figure 4.5: Graphs showing results of Experiment 3
4.4. Experimental Results

**Experiment 4:** Given a plan in a CASO program with multiple subgoals, I calculated the time taken to find the optimum plan among the choices by fixing the look-ahead depth, the number of subgoals, number of variables, the number of constraints per plan and the same objective function and varied the branching factor. Note that for this experiment, I generated a number of CASO programs with different set of plans. I set look-ahead depth = 3, the number of subgoals=3, number of variables =5 and number of constraints/plan = 2.

![Graphs showing results of Experiment 4](image)

**Experiment 5:** Given a plan in a CASO program with multiple subgoals, I calculated the time taken to find the optimum plan among the choices by fixing the branching factor, number of constraints per plan, the look-ahead depth, the number of variables for a given objective function and varying the number of subgoals per plan (size of the plan). I set branching factor = 3, look-ahead depth =3, number of constraints/plan = 2 and number of variables =5.
4.4. Experimental Results

Figure 4.7: Graphs showing results of Experiment 5

Figures 4.3 to 4.7 depict the results for each of the above experiments. As we can see that with increasing depth for the same set of plans, the time taken to find the optimal plan increases. Similar trend is noted when the number of number of variables or constraints per plan is increased although the difference is not that significant as with varying depth. Finally, if branching factor is increased the time taken to find the optimal plan increases as more combinations have to be generated. It is to be noted here though that for every run, I am generating a different set of plans (a different CASO program), solving LP for each of these programs is quite different each time and there is no consistency among them. Thus, for some CASO programs, it might be such that finding a solution may be faster with a higher branching factor than one with a lower factor. For this reason, I randomly generated 100 CASO programs with different branching factors keeping all other parameters constant and found that on an average, with an increase in the branching factor, the time taken to find the optimal
plan also increases.

Overall we can see that the times take (in ms.) are quite small and we can select an optimal very quickly using ECLiPSe together with Jason in my implementation of CASO.
Chapter 5

Extending CASO with User Preferences

In this chapter I extend CASO to incorporate decision making strategies based on user preferences which could be input into the system in real-time.

5.1 Introduction

For many applications there is the need to handle user preferences and customize agents according to the user’s specific needs. It is convenient to let the user provide elaborate specification consisting of constraints, preferences and objectives. Then, let the agent system make decisions about its actions by taking into account changes in the surrounding environment as well as the user preferences that come in real-time. In this chapter I extend CASO to incorporate preferences into the BDI framework.

Let us recall the traveling salesman problem from chapter 3. In our example, we had a salesman trying to reach various destinations with constraints on time and cost. The initial objective was to minimize the traveling cost. For continuity, we again list
the 3 plans to reach destination X below

\[\begin{align*}
\text{p1: } &+!\text{reach}(X) : \text{location}(Y) \& \text{bus}(X,Y,Fare) \& (X!Y) \\
&\quad \leftarrow \text{pay}(Fare); \; !\text{board}_{-}\text{bus}() \\
\text{p2: } &+!\text{reach}(X) : \text{location}(Y) \& \text{taxi}(Fare,Distance) \& (X!Y) \\
&\quad \& \text{dist}(X,Distance) \leftarrow \text{pay}(Fare); \; !\text{board}_{-}\text{taxi}() \\
\text{p3: } &+!\text{reach}(X) : \text{location}(Y) \& \text{train}(X,Y,Fare) \& (X!Y) \\
&\quad \leftarrow \text{pay}(Fare); \; !\text{board}_{-}\text{train}() \\
\end{align*}\]

If all the above three plans are relevant and applicable, then the CASO agent would choose the one which gives the best solution to the objective \(\text{minimize}(\text{Fare})\). However, there may be a reason for the salesman to have preference towards going by the taxi (plan p2) despite having to pay the maximum fare because he may have been tired walking too much. This new preference is not be known at the start of the journey but may have taken place after the salesman has already started his journey. Moreover, this preference may again change while he is on his way.

In this chapter, I extend the BDI architecture of CASO to incorporate the dynamic addition and deletion of preferences. Our work here especially focuses on the use of soft constraints in an agent environment where we give a quantitative dimension to this agent deliberation process by apply c-semiring based techniques to determine the preferred solution. Each agent is given the mandate to achieve defined goals. To do this, it autonomously selects appropriate actions, depending on the prevailing conditions in the environment, based on its own capabilities and means until it succeeds, fails, needs decisions or new instructions or is stopped by its owner. Thus decision agents can be designed to provide interactive decision aids for end-users by eliciting their preferences and then recommending matching products.
BDI [64] agent-oriented systems are flexible and responsive to the environment, and well suited for complex applications with real-time reasoning and control requirements [68]. However, not much work has been done regarding the practical implementation of BDI languages that incorporate user preferences into the BDI framework. These preferences could be modeled as either hard constraints (constraints that must be satisfied with explicit objectives like maximise or minimize *cost*) as in CASO, or soft constraints (constraints that the user would *like* to satisfy). I call this language BAOP (*BDI* Agent with *O*bjectives and *P*references). This work focuses on practical means-ends reasoning which deals with what actions to perform and how to perform the actions. Here I implement BAOP by extending CASO and incorporating a mechanism by which user preferences can be added into the system and also describe a method by which preferences and objectives can be integrated.

### 5.2 BAOP: Extending CASO

Informally, an agent program in BAOP consists of a set of beliefs $\beta$ which includes a set of constraints, a set of objective functions $\Theta$, a set of user preferences $\mu$, a set of events $E$, a plan library $P$, a set of intentions $I$, an objective store $OS$ and a preference store $PS$. There are three selection functions $S_E, S_P, S_I$ to select an event, a plan and an intention respectively. There is also a look-ahead parameter $n$ which determines how many steps the $s$ agent is going to look ahead before committing to a plan or intention. The following sections describe the CASO extensions.

#### 5.2.1 Set of plans ($P$)

**Definition 7.** A BAOP plan $p$ is of the form $t[\varepsilon]:: b_1 \land b_2 \land \cdots \land b_n \land c_1 \land c_2 \land \cdots \land c_m \leftarrow s_{g_1}, s_{g_2}, \cdots, s_{g_k}$ where $t$ is the trigger; $\varepsilon$ refers to the effect of the plan; each $b_i$ refers
5.2. BAOP: Extending CASO

to a belief; each $c_i$ is an atomic constraint; each $sg$ is either an atomic action or a subgoal.

An effect is the result of an activity being executed by some cause or agent. An activity can cause many effects, and an effect can be caused many activities. It should be noted that since CLP assumes Horn Clause, the effects can be Horn Clauses only as we use the effects together with $\beta$ as we will see later. Considering our TSP example, the modified plans in BAOP would be as follows:

\[
\begin{align*}
p1: &+!\text{reach}(X) \ [\text{reachedByBus}(X)]: \\
&\text{location}(Y) & \& \text{bus}(X,Y,Fare) & \& (X!=Y) \\
&<- \text{pay}(Fare); \ !\text{board\_bus}() \\
p2: &+!\text{reach}(X) \ [\text{reachedByTaxi}(X)]: \\
&\text{location}(Y) & \& \text{taxi}(Fare,Distance) & \& (X!=Y) & \& \text{dist}(X,Distance) \\
&<- \text{pay}(Fare); \ !\text{board\_taxi}() \\
p3: &+!\text{reach}(X) \ [\text{reachedByTrain}(X)]: \\
&\text{location}(Y) & \& \text{train}(X,Y,Fare) & \& (X!=Y) \\
&<- \text{pay}(Fare); \ !\text{board\_train}() \\
\end{align*}
\]

5.2.2 Set of user preferences ($\mu$)

The user preference on agent plans can be described through a pairwise comparison of plans. Given two plans in a collection, it is assumed that a user is able to decide if one plan is more useful or relevant than the other plan. Let $P$ denote a finite set of plans.

The user preference may be formally defined by a binary relation $\succ$ on $P$: for $p, p' \in P$, $p \succ p' \iff$ the user prefers $p$ to $p'$. 
5.2. BAOP: Extending CASO

Here $\mu$ is the set of preferences that the agent would like to achieve. Along with objectives, this is yet another natural way of representing softgoals. The preference is given using semiring values and is of the form $<\varepsilon,v_1>$ which depicts a preference value $v_1$ for pursuing the plan. $\varepsilon$ denotes the cumulative effect of plan (or plans).

The fuzzy semiring structure is given be $<A = [0, 1], \oplus = \max, \otimes = \min, 0 = 0, 1 = 1>$ where the preferences between 0 and 1 and higher values denote better preferences. Here 0 is the minimum preference 1 is the maximum preference and the combination is taking the smallest value. In our example user preference are given as fuzzy instance of semiring and are shown in the table below:

<table>
<thead>
<tr>
<th>Effect</th>
<th>Semiring value(0 to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reachedByBus(X)</td>
<td>0.8</td>
</tr>
<tr>
<td>reachedByTrain(X)</td>
<td>0.7</td>
</tr>
</tbody>
</table>

5.2.3 Set of events ($E$)

$E$ is the set of events which now consist of the addition or deletion of user preferences beside the usual set of external and internal events as in CASO.

Thus, a new event would be of the form $(\text{reachedByTaxi}(X), 0.9)$ which means the salesman has a preference value of 0.6 associated with traveling by taxi. If there was no user preference associated with $\text{reachedByTaxi}(X)$ (as is the case here) then this is a new user preference. On the other hand, if the event is of the form $(\text{reachedByBus}(X), 0)$, then it means that minimum preference is associated with $\text{reachedByBus}(X)$ (deletion of user preference).
5.2.4 Preference Store (PS)

PS a consistent set of user preferences and is updated in case a new preference comes in as an event.

Definition 8. A PS is inconsistent if there exists at least two tuples whose conditions are logically equivalent but whose associated semiring values are different.

The machinery I provide ensures such inconsistencies do not occur. Updating of the preference store is formally defined as:

Definition 9. Given a preference store PS and a new objective preference tuple p defined as <ε,v₁>, the result of augmenting PS with p, denoted by PS⁺p, is defined as Υ(MaxPref((PS ∪ p)) where Υ is a choice function and MaxPref(X) is the set of all x ⊆ X such that x is consistent and there exists no x' such that x ⊂ x' ⊆ X and x' is consistent.

5.2.4.1 Preference Store Consistency

The consistency of user preferences in the preference store is maintained by the following logic:

1. Compare new preference tuple <ε,v₁> with existing tuple ε₁ in PS.

2. If <ε,v₁> is logically equivalent to ε₁, replace the value of ε₁ with v₁ else insert the new tuple in PS.

In our example, the preference store would consist of the set of preferences along with their values as given in the table earlier. In case the new event is of the form reachedByTaxi(X), 0.9) as described earlier, then the preference store would now consist of the following:
5.2. BAOP: Extending CASO

<table>
<thead>
<tr>
<th>Effect</th>
<th>Semiring value (0 to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reachedByBus(X)</td>
<td>0.8</td>
</tr>
<tr>
<td>reachedByTrain(X)</td>
<td>0.7</td>
</tr>
<tr>
<td>reachedByTaxi(X)</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Now suppose the new event is of the form \((\text{reachedByBus}(X), 0.6)\). As per the algorithm, consistency check would be applied on the preference store. Since there is already the tuple \(< \text{reachedByBus}(X), 0.8 >\), the store would be updated with the new value as in the following table.

<table>
<thead>
<tr>
<th>Effect</th>
<th>Semiring value (0 to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>reachedByBus(X)</td>
<td>0.6</td>
</tr>
<tr>
<td>reachedByTrain(X)</td>
<td>0.8</td>
</tr>
<tr>
<td>reachedByTaxi(X)</td>
<td>0.9</td>
</tr>
</tbody>
</table>

5.2.5 Option Selection Function \((S_O)\)

**Definition 10.** Given a trigger \(t\) and a set of applicable plans \(\text{AppPlans}(t)\) for \(t\), BAOP option selection function \(S_O\) selects a plan \(p \in \text{AppPlans}(t)\) if and only if: \(p \leq_{\text{opt}} p_i\) for all \(p_i \in \text{AppPlans}(t)\) based on the belief \(\beta\), the objective store \(OS\), the preference store \(PS\) and the goal-plan tree parameter \(n\).

In order to understand the mechanism for selecting the best plan let us consider an example. Let us consider we have two applicable plans - P1 and P2. In order to determine which plan to choose the agent generates the goal-plan tree for all possible paths. The parameter \(n\) creates the pseudo-leaf nodes and therefore we get distinct paths from root to these pseudo-leaf nodes. Figure 5.1 shows all the possible paths from root to pseudo-leaf nodes for the set of plans P1 to P10. The value of \(n\) is 2 which means the goal-plan tree is expanded up to 2 levels.
5.2. BAOP: Extending CASO

Plan1: \(+!t[\varepsilon_1] : Context_1 \leftarrow SG_1; SG_2\).

Plan2: \(+!t[\varepsilon_2] : Context_2 \leftarrow SG_3; SG_4\).

Plan3: \(+!SG_1[\varepsilon_3] : Context_3 \leftarrow a_1\).

Plan4: \(+!SG_1[\varepsilon_4] : Context_4 \leftarrow a_2\).

Plan5: \(+!SG_2[\varepsilon_5] : Context_5 \leftarrow a_3\).

Plan6: \(+!SG_2[\varepsilon_6] : Context_6 \leftarrow a_4\).

Plan7: \(+!SG_3[\varepsilon_7] : Context_7 \leftarrow a_5\).

Plan8: \(+!SG_3[\varepsilon_8] : Context_8 \leftarrow a_6\).

Plan9: \(+!SG_4[\varepsilon_9] : Context_9 \leftarrow a_7\).

Plan10: \(+!SG_4[\varepsilon_{10}] : Context_{10} \leftarrow a_8\).

Broken line (- - -) refers to AND nodes and Solid line (——) refers to OR nodes. 

*Context* \(_i\) is the context of Plan*\(_i\)*; *SG* \(_i\) is subgoal for Plan*\(_i\)*; *a* \(_i\) is an atomic action for Plan*\(_i\)*; *\(\varepsilon_i\)* is the effect of Plan*\(_i\)*;

<table>
<thead>
<tr>
<th>Path id</th>
<th>Possible Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plan1-Plan3-Plan5</td>
</tr>
<tr>
<td>2</td>
<td>Plan1-Plan4-Plan5</td>
</tr>
<tr>
<td>3</td>
<td>Plan1-Plan3-Plan6</td>
</tr>
<tr>
<td>4</td>
<td>Plan1-Plan4-Plan6</td>
</tr>
<tr>
<td>5</td>
<td>Plan2-Plan7-Plan9</td>
</tr>
<tr>
<td>6</td>
<td>Plan2-Plan8-Plan9</td>
</tr>
<tr>
<td>7</td>
<td>Plan2-Plan7-Plan10</td>
</tr>
<tr>
<td>8</td>
<td>Plan2-Plan8-Plan10</td>
</tr>
</tbody>
</table>

Figure 5.1: Agent Plans and corresponding goal-plan tree
5.2.5.1 Multiple Effect Scenarios

Let us now consider any given path (say Path 1) in our earlier example. I follow [11] to discuss the issue of *multiple effect scenarios* in this context. The cumulative effect of this path is not merely a conjunction of the effects of Plan1, Plan3 and Plan5. The effects of each plan are accumulated into cumulative effect annotations in a context-sensitive manner, such that the cumulative effect annotations associated with any plan would describe the effects achieved by the execution of plans were it to execute up to that point. Since we are trying to find out what would be the final effect if we took Path 1 we use multiple effect scenarios as we cannot find the result deterministically.

An *effect scenario* at a given point in a path is one consistent set of cumulative effects of a process if it were to execute up to that point. This is because we may arrive at a given point through multiple paths which cannot be predicted at design time and also activities in a path might undo the effects of activities earlier in the path.

Let \(<P_i, P_j>\) be an ordered pair of plans connected via a sequence flow such that \(P_i\) precedes \(P_j\), let \(\varepsilon_i\) be an effect scenario associated with \(P_i\) and \(\varepsilon_j\) be the immediate effect annotation associated with \(P_j\). Let \(\varepsilon_i = c_{i1}, c_{i2}, \ldots , c_{im}\) and \(\varepsilon_j = c_{j1}, c_{j2}, \ldots , c_{jn}\). If \(\varepsilon_i \cup \varepsilon_j\) is consistent, then the resulting cumulative effect, denoted by \(acc(\varepsilon_i, \varepsilon_j)\), is \(\varepsilon_i \cup \varepsilon_j\). Else, we define \(\varepsilon'_i \subseteq \varepsilon_i\) such that \(\varepsilon'_i \cup \varepsilon_j\) is consistent and there exists no \(\varepsilon''_i\) such that \(\varepsilon'_i \subset \varepsilon''_i \subseteq \varepsilon_i\) and \(\varepsilon''_i \cup \varepsilon_j\) is consistent. I define \(acc(\varepsilon_i, \varepsilon_j) = \varepsilon'_i \cup \varepsilon_j\). We note that \(acc(\varepsilon_i, \varepsilon_j)\) is non-unique as there are multiple alternative sets that satisfy the requirements for \(\varepsilon_i\). Thus the cumulative effect of the two plans consists of the effects of the second plan plus as many of the effects of the first plan as can be consistently included. We remove those clauses in the effect annotation of the first plan that contradict the effects of the second plan. The remaining clauses are undone, i.e., these effects are overridden by the second plan.

Each of these effects may have semiring value associated with them and hence
we get the semiring value for the cumulative effects. If $PS$ contains $(\varepsilon_p, v_p, \varepsilon_q, v_q)$ then the semiring value of $acc(\varepsilon_p, \varepsilon_q)$ would be $v_p \otimes v_q$ where $\otimes$ is the semiring combination operator. Thus from each of the paths, a particular effect scenario is chosen and we get a semiring value for the cumulative effect at each pseudo leaf.

Consider the example of decision making in biomass supply chain from chapter 6. The options for cutting biomass are as whole tree, sticks, chips, billet, rounded and square bales as outlined in Stage 1 Plans (section 6.3.1.1). Again, the options for transportation of the different types of biomass are given in Stage 2 Plans (section 6.3.2). In the new BAOP model, each of the plans related to these options would have effects. The partially filled table below gives the set of effects for each of these options together with a fuzzy semiring value.

<table>
<thead>
<tr>
<th>EffectId</th>
<th>Effect</th>
<th>Semiring value(0 to 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>cutAsWholeTree</td>
<td>0.5</td>
</tr>
<tr>
<td>E2</td>
<td>cutAsChip</td>
<td>0.7</td>
</tr>
<tr>
<td>E3</td>
<td>cutAsBillet</td>
<td>0.6</td>
</tr>
<tr>
<td>E4</td>
<td>cutAsRoundedBales</td>
<td>0.8</td>
</tr>
<tr>
<td>E5</td>
<td>cutAsSquareBales</td>
<td>0.8</td>
</tr>
<tr>
<td>E6</td>
<td>cutAsStick</td>
<td>0.7</td>
</tr>
<tr>
<td>E7</td>
<td>ChipsTransportedOnHighSided</td>
<td>0.6</td>
</tr>
<tr>
<td>E8</td>
<td>ChipsTransportedOnBinsDeck</td>
<td>0.8</td>
</tr>
<tr>
<td>E9</td>
<td>BilletsTransportedOnHighSided</td>
<td>0.7</td>
</tr>
<tr>
<td>E10</td>
<td>WholeTreeTransportedOnDeck</td>
<td>0.6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Thus the cumulative effect of the sequence two plans - one for cutting as chips and the other one for transporting them - is given by $acc(E2, E7)$ or $acc(E2, E8)$. Thus
in this simple example, we consider the cumulative effect of the two plans in sequence to take a decision on which plan to pursue which would be the conjunction of the two effects.

### 5.2.5.2 Handling of Objectives and Preferences

The next item to consider is the OS where we have the set of consistent objective functions that the agent wants to pursue. At each pseudo leaf of the goal plan tree we have

- $\beta$ and context of all plans in the path.
- Semiring value for the effect scenarios.
- Objective function $O$.

There are several choices now. First, the CLP uses CSOP techniques to find the value of the objective function $O$ for each pseudo leaf. Second, the semiring value $v$ of cumulative effect could be considered for each of these leaves. The leaf to be considered would be the one which has both these values as the highest. It is a policy that the user can decide beforehand to give priority to either the objective or the semiring value as these values are nothing but natural means of representing softgoals that the agent would like to achieve. Next we choose the path which has the maximum value at the pseudo leaf based on the strategy chosen.

### 5.2.6 Intention Selection Function ($S_I$)

$S_I$ function selects one of the agent’s intentions, i.e., one of the independent stacks of partially instantiated plans within the set of intentions by applying techniques similar
to that of $S_O$. Each intention stack is a choice for the agent. In order to determine right intention, the agent first considers the top element (i.e., a partially instantiated plan) of every intention stack. Each of these intentions for a goal plan tree and here also we solve the CSOP which consists of $\beta$, $OS$, $PS$ and $n$. As before, we have several choices and we can apply different strategy to choose a particular intention.

5.3 BAOP interpreter

The BAOP interpreter is similar to the CASO interpreter with main difference being the addition of preference store and is shown in figure.

The BAOP interpreter matches the context of each plan with the external triggering event and this generates a set of all relevant plans. The context part of the plans is unified against the agents beliefs and Constraint solving is now performed on these relevant plans to determine whether the constraint(s) in the context of the plan is (are) consistent with the belief base. This results in a set of applicable plans (plans that can actually be used at that moment for handling the chosen event). $S_O$ now chooses the best applicable plan based on both the Objective Store and the Preference Store.

At every interpretation cycle of an agent program, BAOP updates a list of events, which may be generated from perception of the environment, or from the execution of intentions. Beliefs are updated from perception and whenever there are changes in the agents beliefs, this implies the insertion of an event in the set of events. On top of the selected intention there is a plan which would be chosen for execution. This implies that either a basic action is performed by the agent on its environment, an internal event is generated (in case the of an achievement goal or a test goal is performed. If the intention is to perform a basic action or a test goal, the set of intentions needs to be updated. In the case of a test goal, the belief base will be searched for a belief atom
that unifies with the predicate in the test goal. If that search succeeds, further variable instantiation will occur in the partially instantiated plan which contained that test goal (and the test goal itself is removed from the intention from which it was taken). In the case where a basic action is selected, the necessary updating of the set of intentions is simply to remove that action from the intention and initialize the parameters based on the solving the relevant CSOPs as well as consulting the Preference Store. When all formulae in the body of a plan have been removed (i.e., have been executed), the whole plan is removed from the intention, and so is the achievement goal that generated it

Figure 5.2: BAOP execution cycle
(if that was the case). This ends a cycle of execution, and BAOP starts all over again, checking the state of the environment after agents have acted upon it, generating the relevant events, and so forth.

5.4 Related Work

Many of the other BDI languages have been extended to incorporate constraints and preferences to guide the agent to select the best possible plan.

In Jason plan rules can feature a plan label. This label is noted after the plan rule’s Head and can take the form of a predicate, helping the interpreter choose one plan over another. For example, it can contain information about the plan, such as its cost or payoff. However, the plan label with cost or payoff has to be given by the programmer at design time and there is no way to change the payoff once the interpreter starts running. Therefore, the same plan with highest payoff would be selected in most of the cases.

Plan selection by preferences is quite different in our case as the preferences can be changed during run-time thereby enabling the best plan to be selected based on preference.

In [32] a system that allows the user to express all these kinds of constraints and preferences is described by extending GOLOG [50]. The authors address the problem of combining non-Markovian qualitative preferences, expressed in first-order temporal logic, with quantitative decision-theoretic reward functions and hard symbolic constraints in agent programming. In another approach [69], prioritized goals have been integrated with the IndiGolog agent programming language.

In our approach the semiring specifications are richer than their specification of prioritized goals. Also, I combine both qualitative and quantitative constraints.
In [40] an approach has been made to extend GOAL (a language based on propositional logic) with temporal logic for the representation of goals and preferences for additional expressiveness. In this approach hard and soft constraints have been integrated into it as well as as achievement goals, maintenance goals, and temporally extended preferences. Here if an agent has the ability to lookahead a number of steps, it can also use its goals to avoid selecting those actions that prevent the realization of (some of) the agents goals. Effects of basic actions are taken into account which change the mental state of agents. The work is based on the fact that planners could be guided to select preferred plans.

One of the major differences with regards to our work is that the preferences and objectives are dynamic - they work like belief updates when the user changes his mind. Also, unlike other approaches, the hard constraints that I talk about here are objective functions in the current environment and not the agent goals.

In another approach [10], the authors describe a model where preferences and constraints over goals can be specified. They also mention an algorithm PREFPLAN for solving the resulting constrained optimization problem. PDDL3 [34] is an action-centred language in the planning domain where strong and soft constraints on plan trajectories (possible actions in the plan and intermediate states reached by the plan), as well as strong and soft problem goals has been incorporated.

One of the differences between agent programming and planning is that in planning one compares complete plans against the available constraints, while in agent programming the constraints are into account continuously during execution. In this context, the above frameworks deal mainly with plan uncertainty whereas our approach has been to incorporate similar notions into BDI agent paradigm.

There are a number of frameworks which mix planning and BDI-style execution. Of particular interest is CANPLAN [67] which have built-in capacity for lookahead.
In contrast, our lookahead mechanism is domain independent and is built into the BDI architecture. Moreover, when to do a lookahead is dependent on the user and in a highly reactive system, full lookahead may not give the best possible outcome.
Chapter 6

CASO Case Study: Decision support in a biomass supply chain

One of the challenging issues in a distributed time sensitive information rich environment is how to assist the decision makers to make decisions quickly and effectively. In this chapter I show how CASO can be used in Biomass Supply Chain Management. The case study shows how the introduction of optimization using agent technology improves the quality of decision support in real-life situations.

6.1 Biomass Supply Chain

Biomass is the name given to all plants and animals on earth. Energy from biomass refers to ways of using plants and animals as energy sources. There are a number of technological options available to make use of a wide variety of biomass types as a renewable energy source as described below.

- directly by producing electricity, normally done by burning the biomass (e.g. wood or waste products) in steam generators.
indirectly by converting it into a liquid or gas fuel.

Biomass Supply Chains (BSCs) for energy production are comprised in general of four general system components:

1. biomass harvesting / collection (from single or several locations) and pre-treatment
2. storage (in one or more intermediate locations)
3. transport (using a single or multiple echelons)
4. final conversion

Traditionally, biomass has been used for energy (mainly thermal) production in areas close to its production sites. However, an emerging practice for energy producers is to procure biomass from several suppliers in order to develop the critical mass necessary for the justification of an energy production facility. The increased complexity of this system dictates the need for developing sophisticated customized supply chain planning and coordination methodologies as opposed to the well explored traditional supply chain management.

A wide range of options exists for the various components of the Biomass Supply Chain (BSC). Selection of each component in the chain often impacts on other components. The supply chain considerations and costs of using biomass fuel on a large scale for electricity generation at power stations is quite complex and is made up of a range of different activities. This scenario is adapted from [2]. The activities can include ground preparation and planting, cultivation, harvesting, handling, storage, in-field or forest transport, road transport and utilisation of the fuel at the power station. Moreover, the options for supplying the end user with biomass fuel of the right
specification, in the right quantity at the right time from resources which are typically diverse and often seasonally dependent. Moreover, the transport infrastructure is usually such that road transport will be the only potential mode for collection of the fuel. The components described above could have all possible combinations and permutations and some of the actions may not be necessary at all. Each component is optimized for economic costs and energy costs for various parameters that determine its structure and performance, that is, for production (scale, location and biomass type); for harvesting (technique and biomass type); for storage (technique, location and biomass type); for pretreatment and densification (scale, technique, biomass type) and for transporting (stage, mode, distance and bioenergy type).

6.2 Decision Making and Optimisation in BSC

In order to supply biomass from its point of production to a power station several decisions and activities need to be taken:

1. What is the source of biomass? Sources of biomass, like residues of the agric and horticulture; residues of the food industry; import and multi-functional cultivation.

2. Where to dry and store the biomass? Many types of biomass will be harvested at a specific time of year in a headland but will be required at the power station on a year round basis, it will therefore be necessary to store them. The storage point can be located on the headland or in the farm, at the power station or at an intermediate site.

3. How to load and unload road transport vehicles? Loading and unloading of road transport vehicles has to be done efficiently to minimize wastage. Typically the crops are cut into sticks, billets, chips or bales and loaded onto the vehicles.
4. *How to transport to power station?* Transport by road transport vehicle using heavy goods vehicles for transport to the power station is used due to the average distance from farms to power station, and the carrying capacity and road speed of such vehicles.

5. *How to process the biomass?* Processing of the biomass to improve its handling efficiency and the quantity that can be transported.

From the above it is quite clear that decisions need to be taken at various decision points - some of these could be taken beforehand during the planning stage (questions 1, 2 and 5) whereas others (questions 3, 4) need to be taken *impromptu*. The latter ones could include the choice of the type of vehicle which may be affected by the weather and road conditions or the type of equipment to cut the branches.

The BSC is made up of heterogeneous production subsystems gathered in vast dynamic and virtual coalitions. It is to be noted that multiagent systems enable increased autonomy of each member in the supply chain. Each production subsystem pursues individual goals while satisfying both local and external constraints. Therefore, one or more agents can be used to represent each subsystem in the BSC where each agent would be responsible for vagarious decision makings.

### 6.2.1 Harvesting and Handling of Biomass

The different components of harvesting and handling supply chain are closely interconnected [73]. For each of the aspects of supply chain given below, the interactions affecting other components has to be taken into account. Since these are carried out each day, decisions may be changing each time because of several factors relating to weather, availability of machines, trucks, road conditions etc.
• Overall harvester machine size and design concept.

• Harvester capacity and productivity (ha/hour, overall work rates).

• Width of cut.

• Cutting mechanism.

• Comminution (breaking of Biomass into smaller pieces).

• Transfer mechanism of biomass material from harvester to the next step in the supply chain.

• Transport from field to storage (intermediate or on-farm).

• Road transport to plant.

• Storage, if required.

• Drying, if required.

6.2.1.1 Transportation of Biomass

Let us consider a snapshot of BSC where we look into the method of transporting the biomass from field to storage. The various types of trailers available are:

1. flat bed or forestry trailer for stems, sticks or bales

2. bulk commodity trailer for chips/billets

3. trailer fitted with grapple for compacting the load for whole trees

If we now consider the various transport options of these to the conversion plant several things need to be considered. These include:
• the average distance travelled to the conversion plant site

• transport regulations regarding axle weights and vehicle widths and lengths

• road classifications for the proposed route to be taken

• the road width and traffic density

• handling equipment available at the collection and delivery points

• the need to maximise payloads to minimise costs

• the maximum height of loads and length of truck-trailer combinations.

• the type of vehicle for carrying the different type of biomass (sticks, chips or bales) - bulk densities for different forms of biomass feedstock vary considerably, which impacts on the choice of transport.

6.2.1.2 Storage of Biomass

There are several factors to be considered in terms of the type and size of storage facilities, as well as storage and preservation methods. The various considerations include

• storage capacity

• storage and preservation methods such as ensiling, chemical preservation, drying, or freezing

• effect do moisture content, composition and quality of biomass material
6.2.2 Optimisation considerations

The optimal choice of a biomass type, type of power plant, locations, transport, storage and pre-treatments needs to be made at the right time. Here both financial and energetic goals need to be addressed. The returns should be enough to cover the costs and the energy used for the logistics should be much lower than the energy returns in the plant.

There are several competing objectives which needs to be addressed. These are social, economic as well as environmental which are often competing amongst themselves.

**Social objective**: Minimize the number of tractors loaded with crop going through small village and other sensitive areas to reach the plant

**Transportation objective**: Minimize the unit costs of road transport loss

**Storage objective**: Minimize chip storage time in case of chipping at headlands (losses are assumed to occur at a constant rate of 4 per cent per month)

**Environmental Objective**: Minimize environmental emissions and sound (use battery-powered machinery)

**Economic Objective**: Minimize cost of transport and labour

As one can see, the *economic* objective is in conflict with *environmental* as using battery-powered equipment may be more expensive.

6.3 CASO agent in Biomass Supply Chain

The BSC can be represented as a hierarchical planning system where solutions are obtained by decomposing high level tasks (or goals) into more specific tasks and prim-
itive actions. This makes it also suitable to be used as a BDI system where the best solution could be searched for before executing the plan. This is particularly suitable in a dynamic environment like the BSC. CASO is an ideal candidate for use as a BDI agent system in implementing intelligent behaviour in the BSC where there are several choice points. Here only a single CASO agent is described for decision making but the entire BSC can be easily modeled with several CASO agents.

6.3.1 Stage 1: Cutting and Transporting Biomass to Storage

For simplicity, let us consider only the cutting and transportation part of the BSC and show how a CASO agent could be used to take decisions. The agent is deployed on a PDA which would assist the field supervisor situated in the harvesting field to take decisions. It has several plans at its disposal and can choose one of the possible alternate plans based on the current set of constraints. The top level goal of the agent is to decide which type of cutting and the subsequent comminution of biomass is to be conducted based on the type of machine available for cutting and the trailer available for transport to the storage. The type of machine and trailer available and the cost associated with using each of these are stored as agent beliefs. The possible options to cut and transport the biomass to an intermediate storage are given below:

1. cut as whole tree which requires machine of type $M_1$ and trailer fitted with grappler
2. cut as sticks which require machine of type $M_2$ to cut and either forestry or flatbed trailer
3. cut tree and produce chips which requires machine of type $M_3$ and bulk commodity trailer
4. cut tree and produce *billet* which requires machine of type $M_4$ and forestry trailer

5. cut tree and produce *rounded bales* which requires machine of type $M_5$ and either forestry or flatbed trailer

6. cut tree and produce *square bales* which requires machine of type $M_6$ and either forestry or flatbed trailer

### 6.3.1.1 CASO Plans for Stage 1

Figure 6.1 lists the set of CASO plans based on the above assumptions. The plans are of the form

```plaintext
+!cut() : tractor(T) & machine(M) <- moveItemToStorage(CostToStorage);
+atStorage(Item); !transportItemToPlant();
```

where

- +!cut() is the top-level goal of cutting and transporting biomass to storage;
- tractor(T) and machine(M) are the plan contexts where T refers to the type of tractor (grappler, forestry etc.) and M refers to the type of machine used ($M_1, M_2$ etc.);
- Item refers to the type of biomass (whole tree, chips etc.) being transported;
- moveItemToStorage(CostToStorage) is an action which refers to the actual movement of the Item and CostToStorage is the action parameter which depicts the amount that needs to be paid for this transportation;
- +atStorage(Item) refers to addition of a belief that Item is at storage;
6.3. CASO agent in Biomass Supply Chain

- `!transport ItemToPlant()` is a sub-plan which describes how the `Item` would be transferred to plant. The subplans are further expanded in figure 6.5 and are described in a later section.

```plaintext
// Top level plans relating to cutting and transportation from field to storage

Plan #1:
+!cut() : tractor(grappler) & machine(M1) ← moveWholeToStorage(CostToStorage);
+atStorage(Whole); !transportWholeToPlant();

Plan #2:
+!cut(): tractor(forestry) & machine(M2) ← moveStickToStorage(CostToStorage);
+atStorage(Stick);!transportStickToPlant();

Plan #3:
+!cut(): tractor(flatbed) & machine(M2) ← moveStickToStorage(CostToStorage);
+atStorage(Stick);!transportStickToPlant();

Plan #4:
+!cut(): tractor(bulkCommodity) & machine(M3) ← moveChipToStorage(CostToStorage);
+atStorage(Chip);!transportChipToPlant();

Plan #5:
+!cut(): tractor(forestry) & machine(M4) ← moveBilletToStorage(CostToStorage);
+atStorage(Billet);!transportChipToPlant();

Plan #6:
+!cut(): tractor(forestry) & machine(M5) ← moveRndBaleToStorage(CostToStorage);
+atStorage(RndBale);!transportRndBalToPlant();

Plan #7:
+!cut(): tractor(flatbed) & machine(M5) ← moveRndBaleToStorage(CostToStorage);
+atStorage(RndBale);!transportRndBalToPlant();

Plan #8:
+!cut(): tractor(forestry) & machine(M6) ← moveSqBaleToStorage(CostToStorage);
+atStorage(SqBale);!transportSqBaleToPlant();

Plan #9:
+!cut(): tractor(flatbed) & machine(M6) ← moveSqBaleToStorage(CostToStorage);
+atStorage(SqBale);!transportSqBaleToPlant();
```

Figure 6.1: Plans related to top level goals of cutting and transporting to storage

6.3.1.2 CASO Beliefs during Stage 1

Let us assume that the following are true at the time when it is required to cut the trees and transport them to the storage:
• Machines $M_1$ and $M_2$ are available

• The total amount of biomass that need to be transported in 100 tons.

• Tractors of type grappler, forestry and flatbed are available with operating cost per hour, maximum trips per hour and maximum capacity of using each vehicle are as shown in table 6.1.

<table>
<thead>
<tr>
<th>Trailer Type</th>
<th>Cost p.h.</th>
<th>Max. Trips p.h.</th>
<th>Max capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>grappler</td>
<td>$25</td>
<td>4</td>
<td>5 ton.</td>
</tr>
<tr>
<td>forestry</td>
<td>$25</td>
<td>4</td>
<td>5 ton.</td>
</tr>
<tr>
<td>flatbed</td>
<td>$25</td>
<td>4</td>
<td>5 ton.</td>
</tr>
</tbody>
</table>

Table 6.1: Rates for different types of trailers

The beliefs for the CASO agent based on the above environmental conditions are written as follows:

trailer(grappler).
trailer(forestry).
trailer(flatbed).
machine($M_1$).
machine($M_2$).

The beliefs trailer(grappler), trailer(forestry), trailer(flatbed) are CLPs which contain information on various rates (as shown table 6.1) and are depicted in figures 6.2, 6.3 and 6.4 respectively.

6.3.1.3 Plan Selection at Stage 1

From the various plans that at the disposal of the CASO agent, there are 3 plans - Plans 1, 2 and 3 are applicable when the new triggering event $!cut()$ comes in the
6.3. CASO agent in Biomass Supply Chain

**trailer(grappler):-**

Vars = [TotalWeight, CostPerHr, TripPerHr, CapacityPerTrip],
CostPerHour #= 25,
TotalWeight #= 100,
TripPerHour #=< 4,
CapacityPerTrip #=< 5,
CostToStorage #= CostPerHr*TotalWeight/(TripPerHr * CapacityPerTrip)

Figure 6.2: Belief(CLP) of CASO Agent related to trailer fitted with grappler

**trailer(forestry):-**

Vars = [TotalWeight, CostPerHr, TripPerHr, CapacityPerTrip],
CostPerHour #= 22,
TotalWeight #= 100,
MaxWorkingHours #= 10,
TripPerHour #=< 6,
CapacityPerTrip #=< 8,
CostToStorage = CostPerHr*TotalWeight/(TripPerHr * CapacityPerTrip)

Figure 6.3: Belief(CLP) of CASO Agent related to forestry trailer

**trailer(flatbed):-**

Vars = [TotalWeight, CostPerHr, TripPerHr, CapacityPerTrip],
CostPerHour #= 21,
TotalWeight #= 100,
TripPerHour #=< 6,
CapacityPerTrip #=< 8,
CostToStorage = CostPerHr*TotalWeight/(TripPerHr * CapacityPerTrip)

Figure 6.4: Belief(CLP) of CASO Agent related to flatbed trailer

event queue. The plans are based on the fact that the context conditions of the plans match those that are there in the belief base namely, machines $M_1$ and $M_2$ and the trailers of type *grappler, flatbed* and *forestry* are available. Let the objective of the agent is to minimize $CostToStorage$.

Plan selection mechanism would now involve solving the CLPs relevant to each of the Plans 1, 2 and 3 and picking up the one which minimizes $CostToStorage$. In this
example, the plan chosen would be Plan 2. Note that the if this plan is chosen to be executed the parameter CostToStorage of the action is instantiated with the actual value. Now, if the objective was to minimize trailer movement (a social objective so that noise pollution is less) then CASO agent would solve the CLPs with the objective minimize(Trips per hour), the plan chosen would be 3. Thus we see the particular plan selected depends on the prevalent conditions and the current objective.

6.3.2 Stage 2: Transporting Biomass from Storage to Plant

Once the items reach storage, the next decision to be taken is how to transport of these to the power plant. Bulk densities for different forms of biomass feedstock vary considerably and this impacts the choice of transport. Note that as with the earlier set, the agent beliefs consist of the availability of vehicles along with their costs. Again, the options available are given below:

1. chips in a high sided vehicle
2. chips in bins on a flat deck truck
3. billets in a high sided vehicle
4. billets in bins on a flat deck
5. whole trees or sticks in a high sided vehicle
6. whole trees or sticks strapped on to a flat deck
7. bales stacked on a flat deck.

6.3.2.1 CASO Plans for Stage 2

Figure 6.5 lists the set of CASO plans based on the above assumptions.
<table>
<thead>
<tr>
<th>Plan #10:</th>
<th>Plan #11:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+!transportWholeToPlant() : truck(highSided) &amp; atStorage(Whole) ← loadTruck($T_{M_1}$, $T_{M_2}$); moveToPlant().</td>
<td>+!transportWholeToPlant() : truck(flatDeck) &amp; atStorage(Whole) ← loadTruck($T_{M_1}$, $T_{M_2}$); moveToPlant().</td>
</tr>
<tr>
<td>Plan #12:</td>
<td>Plan #13:</td>
</tr>
<tr>
<td>+!transportSticksToPlant() : truck(highSided) &amp; atStorage(Sticks) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
<td>+!transportSticksToPlant() : truck(flatDeck) &amp; atStorage(Sticks) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
</tr>
<tr>
<td>Plan #14:</td>
<td>Plan #15:</td>
</tr>
<tr>
<td>+!transportChipToPlant() : truck(flatDeck)&amp; bin() &amp; atStorage(Chip) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
<td>+!transportChipToPlant() : truck(highSided) &amp; atStorage(Chip) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
</tr>
<tr>
<td>Plan #16:</td>
<td>Plan #17:</td>
</tr>
<tr>
<td>+!transportBilletToPlant : truck(flatDeck)&amp; bin() &amp; atStorage(Billet) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
<td>+!transportBilletToPlant : truck(highSided) &amp; atStorage(Billet) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
</tr>
<tr>
<td>Plan #18:</td>
<td>Plan #19:</td>
</tr>
<tr>
<td>+!transportRndBalesToPlant : truck(flatDeck) &amp; atStorage(RndBales) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
<td>+!transportSqBalesToPlant : truck(flatDeck) &amp; atStorage(SqBales) ← loadTruck($T_{M_1}$, $T_{M_2}$); transportToPlant().</td>
</tr>
</tbody>
</table>

Figure 6.5: Plans related to transporting Biomass from Storage to Plant
6.3.2.2 CASO Beliefs during Stage 2

Let us assume that the following are true at the time when it is required to transport Biomass to the plant:

- Flat deck truck which runs on diesel and High sided truck which runs on petrol are available with varying capacities, fuel price and fuel consumption per trip as shown in table 6.1.

- Sticks need to be transported to the plant and the total amount is 100 tons.

<table>
<thead>
<tr>
<th>Truck Type</th>
<th>Fuel Price</th>
<th>Fuel needed per trip</th>
<th>Max capacity per trip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat deck</td>
<td>$1.5</td>
<td>20 lt.</td>
<td>10 ton.</td>
</tr>
<tr>
<td>High sided</td>
<td>$1.6</td>
<td>30 lt.</td>
<td>15 ton.</td>
</tr>
</tbody>
</table>

Table 6.2: Rates for different types of trucks

The beliefs for the CASO agent based on the above environmental conditions are written as follows:

\[
\text{truck(flatDeck), truck(highSided).}
\]

The beliefs truck(highSided), truck(flatDeck) are CLPs which contain information on various rates (as shown table 6.2) and are depicted in figures 6.6 and 6.7 respectively.

6.3.2.3 Plan Selection at Stage 2

From the various plans that at the disposal of the CASO agent, there are 2 plans - Plans 12, 13 are applicable when the new triggering event \( transportSticksToPlant() \) comes
6.3. CASO agent in Biomass Supply Chain

\textbf{truck(\texttt{highSided}):-}
Vars = [TotalWeight, DieselPrice, FuelPerTrip, TruckCapacityPerTrip, Trips],
TotalWeight #= 100,
DieselPrice #= 1.6,
FuelPerTrip #= 30,
TruckCapacityPerTrip #=< 15,
Trips #= TotalWeight/TruckCapacityPerTrip,
CostToPlant #= DieselPrice*FuelPerTrip*TotalWeight/TruckCapacityPerTrip.

Figure 6.6: Belief(CLП) of CASO Agent related to high sided truck

\textbf{truck(\texttt{flatDeck}):-}
Vars = [TotalWeight, PetrolPrice, FuelPerTrip, TruckCapacityPerTrip, Trips],
TotalWeight #= 100,
PetrolPrice #= 1.5,
FuelPerTrip #= 20,
TruckCapacityPerTrip #=< 10,
Trips #= TotalWeight/TruckCapacityPerTrip,
CostToPlant #= PetrolPrice*FuelPerTrip*TotalWeight/TruckCapacityPerTrip.

Figure 6.7: Belief(CLП) of CASO Agent related to flat deck truck

in the event queue. The plans are based on the fact that the context conditions of the
plans match those that are there in the belief base namely, the trucks of type \texttt{flatDeck}
and \texttt{highSided} are available. Again, let the objective of the agent is to minimize
\texttt{CostToPlant}.

Plan selection mechanism would now involve solving the CLПs relevant to each of
the Plans 12, and 13 and picking up the one which minimizes \texttt{CostToPlant}. In this
example, the plan chosen would be Plan 12. As before, if the objective was to minimize
truck movement (a \textit{social objective} so that noise pollution is less) then CASO agent
would solve the CLПs with the objective minimize(\texttt{Trips}), the plan chosen would be
13. Thus we see the particular plan selected depends on the prevalent conditions and
the current objective.
6.3.3 Stage 3: Storage of Biomass at Conversion Plant

Once the items are in the power plant the storage considerations come into place. As mentioned earlier, several factors like moisture content, storage space etc. play a vital role in the decision. As before, we have several beliefs and plans at our disposal at this stage the details of can be worked out in a similar way.

6.3.4 Plan Selection at Stage 1 using 1 and 2-step lookahead

So far what we looked at consists to doing no lookahead. That is, the CASO agent at Stage 1, checks the plans and chooses the optimal one based on some objective. If we want to apply 1-step lookahead, the agent would try to expand the next set of subgoals and try to compute the best path. This means expanding the goal-plan tree to evaluated each path.

As in our example, let the set of beliefs prevalent during Stage 1 hold true. Also, let the beliefs prevalent at Stage 2 also hold true at Stage 1. In other words, it is believed that both high sided and flat deck trucks would be available at storage. Also, let us assume that the objective is to minimize the overall cost. Therefore the possible paths based on the applicable plans are as shown in table 6.3. The right path to choose at Stage 1 using lookahead would encompass solving the CSOPs along the path and choosing the one which would produce the optimal objective which is the sum of the costs associated with transferring the biomass from field to storage and then from storage to plant. Thus with 1-step lookahead, the agent is better informed and therefore able to choose the right plan.

The example is carried out using 1-step lookahead but the notion could easily be extended further to incorporate more than 1-step. If we consider the stage 3 activities,
6.3. CASO agent in Biomass Supply Chain

<table>
<thead>
<tr>
<th>Option id.</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Plan1-Plan10</td>
</tr>
<tr>
<td>2</td>
<td>Plan1-Plan11</td>
</tr>
<tr>
<td>3</td>
<td>Plan2-Plan12</td>
</tr>
<tr>
<td>4</td>
<td>Plan2-Plan13</td>
</tr>
<tr>
<td>5</td>
<td>Plan3-Plan12</td>
</tr>
<tr>
<td>6</td>
<td>Plan3-Plan13</td>
</tr>
</tbody>
</table>

Table 6.3: Possible paths

then we can apply 2-step lookahead to find the optimal plan. While the above process of determining the best plan through look-ahead technique helps the supervisor to choose the best plan in the current circumstance there could be change in beliefs or objective once Stage 1 is executed. Let the new objective to be considered is the minimization of number of trips. This is a social objective which requires less number of trucks plying through the villages. The objective store now contains minimize (Cost) and minimize(Trips). These two objectives are in conflict with each other since it may be more expensive to make less trips using a bigger truck than to make more trips on a smaller truck. The objective store now checks for this conflict and chooses the later one and discard the earlier one if they are not consistent. This may result in another plan being selected now instead of the plan which may have been chosen if all conditions would have remained the same.
Chapter 7

Constraint based Optimizations in other BDI languages

In this chapter I discuss how the CASO notion of applying objectives and optimization techniques can be easily incorporated in some other representative BDI languages like 3APl, 2APL and CAN.

7.1 Introduction

BDI languages incorporate the autonomous behaviour and provide sophisticated mechanisms for controlling, and reasoning about such behaviours. There are many programming languages based on the BDI approach as we have seen earlier which are very popular. The languages are similar in nature but are still different as they are not based on agreed notions.

For all BDI based languages both the actual behaviours (the plans) and the situations for which they are appropriate (their context conditions) are fixed at design time. Because of this, it is often difficult or impossible for the programmer to craft
the exact conditions under which a plan would succeed. Also, once deployed, the plan selection mechanism is fixed and may not adapt to potential variations of different environments. The plan selection mechanism that has been defined in CASO in earlier chapters can be easily applied to these languages which would help to eliminate the above limitations in these languages for selecting plans and intentions dynamically.

7.2 3APL

3APL [38] is a logic based programming language for implementing cognitive agents that follows the classical BDI architecture. The semantics of 3APL agents is defined by a transition system composed of transition rules. The use of 3APL provides the agent programmer with a very intuitive and simple way to define agents. The programmer can declaratively specify agents beliefs (represented by Horn Clauses) and goals (represented by conjunctions of atoms), how they build plans to achieve such goals, and reason with their beliefs. Beliefs are a set of Prolog rules and facts, which can additionally be complemented by beliefs which are imported from an external Prolog program when the agent is initiated. Furthermore, communication between agents can be done in an elegant way by modifying the beliefs of agents, allowing for the possibility of reasoning with the transferred messages. A 3APL agent contains formal definitions of agent beliefs, capabilities, goals and plans. Specifically, there are six skeletal blocks that must be defined as mentioned below:

- BELIEFBASE
- CAPABILITIES
- GOALBASE
- PLANBASE
7.2. 3APL

- PG-RULES
- PR-RULES

Capabilities consist of belief updates, external actions, or communication actions. Initial goals are defined in the goal base. Each goal has associated goal planning rules (PG rules), which serve as abstract plans and are called from the goals as long as their guard conditions are met. The PG rules in turn can call plan revision rules, or PR rules, which serve as subroutines, and can be called upon to execute lower level and/or repetitive tasks as long as their guard conditions are met. Initial plans are defined in the plan base, executed at the beginning of the deliberation cycle.

7.2.1 3APL deliberation cycle

A 3APL agent can execute plans or execute reasoning rules to modify its intentions or plans. However, it is necessary to have a strategy to guide which reasoning rule or plan should be selected to be executed by the agent. In the implementation of 3APL, this strategy is fixed by a deliberation cycle. First, the agent tries to apply an applicable Goal Planning Rule, i.e., a reasoning rule where its precondition is satisfied and that there is a goal in the goal base that triggers this rule. Afterwards it tries to apply an applicable Plan Revision Rule, i.e., a reasoning rule where its precondition is satisfied and there is a plan in the intention base that triggers the rule. Finally, it tries to select a plan and, if it is possible, executes it. There is no criteria for selecting a specific reasoning rule or plan to be executed, it just ‘picks’ one. The deliberation cycle of 3APL agent can be summarized as follows:

1. Find Plan Generation Rules that Match Goals
2. Remove Plan Generation Rules with atoms in head that exist in Belief Base
3. Find Plan Generation (PG) Rules that Match Beliefs

4. Select a Plan Generation (PG) Rule to Apply

5. Apply the Plan Generation (PG) Rule, thus adding a plan to the planbase

6. Find Plan Revision (PR) Rules that Match Plans

7. Find Plan Revision (PR) Rules that Match Beliefs

8. Select a Plan Revision (PR) Rule to Apply to a Plan

9. Apply the Plan Revision (PR) Rule to the Plan

10. Find Plans To Execute

11. Select a Plan To Execute

12. Execute the (first basic action of the) Plan

### 7.2.2 Example

Let us consider an example where an agent can either drink soda or juice from the tap to quench thirst (*agent goal*). Let us assume that the agent also believes that there is glass available as well as water and soda. This example is written below in 3APL but it could be represented equally in 2APL. The 3APL agent contains formal definitions of agent beliefs, capabilities, goals and plan. Formally, the beliefs can be thought of as consisting of the following: current status of the agent i.e, the agent is in standby mode, a glass being present and both juice and soda being available. The status of the agent could change to either drinking soda or drinking water during the lifetime of the agent. The initial beliefs, capabilities, goals, plans, PG-Rules and PR-Rules are described below.
BELIEFBASE {
    status(standby).
    have_glass().
    juice_Available().
    soda_Available().
}

CAPABILITIES {
    {status(S1)} SetStatus(S2)
    {NOT status(S1), status(S2)},
    {drink()},
    {have_juice()},
    {have_soda()},
    {get_soda(),
    {get_juice()}
}

GOALBASE {
    quench_thirst().
}

PG-RULES {
    quench_thirst()<-have_glass() | {
        SetStatus(drinkingSoda());
        have_soda();
    }
    quench_thirst()<-have_glass() | {

SetStatus(drinkingWater());
    have_juice();
}
}

PLANBASE { SetStatus(started); }
PR-RULES {
    have_soda()<-soda_Available() | {
        get_soda(),
        drink();
    }
    have_juice()<-juice_Available() | {
        get_juice(),
        drink();
    }
}

From the above example, it is not clear which plan the agent would choose - whether to go to drink water or soda. The agent would simply pick one up and execute it.

### 7.2.3 Extending 3APL with constraint-based optimization

The main conceptual difference between CASO (or AgentSpeak(L)) and 3APL are the concepts of event and intention. There are no direct counterparts of these in 3APL and these play an important part in option and intention selection mechanisms in CASO. In CASO events can be used to place subgoals in a queue for eventual deliberation by the agent and further planning (all of which may be interrupted by higher-priority
events). The new plan generated for a subgoal is then added onto an existing stack of plans (i.e., the intention) for execution. However, in 3APL, events are not supported since goals are themselves modified in the process of planning and acting; instead of using intentions, 3APL simply attempts to execute its current goals.

Our proposal to extend 3APL consists of the following:

1. Agent belief bases can be represented as CLPs instead of LPs (i.e., Prolog).

2. The ability in AgentSpeak(L) and CASO to introduce events/goals/objectives on the fly could be achieved in 3APL by defining a specially designated User Agent that uses the existing inter-agent messaging facility in 3APL to send messages.

3. Optimization objectives could be introduced as non-triggering events which would be used for plan and intention selection.

4. Plan selection with parametric look-ahead can be constructed by creating the goal-plan tree using PG-rules and PR-rules.

5. Action parameters could be introduced into the plan body which would be instantiated when the plan is to be executed.

7.2.3.1 Example revisited

For the example described earlier, we can apply these principles to choose a particular plan. Let us change the beliefs juice_available and soda_available such that they can be represented as CLP. The new beliefs would now look like as follows:

```
juice_available() :-
    Vars = [juice_cal, cal_intake],
```
7.2. 3APL

```
Vars :: 0..inf,
float(Vars),
juice_cal=1.0,
cal_intake =juice_cal.
```

```
soda_available() :-
    Vars = [soda_cal, cal_intake],
    Vars :: 0..inf,
    float(Vars),
soda_cal=1.8,
cal_intake =soda_cal.
```

3APL can be easily extended to include option and intention selection based on an agent’s objectives. Suppose the objective of the agent is to drink juice or soda based on calorie content. The agent would choose the drink with the least calorie content. Observe that this involves an objective function - that of minimizing calorie intake. This information need to be stored within the agent. In the 3APL, there is no obvious way of doing this. The optimization function could therefore be \texttt{optimize(min(cal_intake))}. Thus when the agent wants to select a plan, it executes each CLP and gets the value of \text{cal_intake}.

If we look closely at the deliberation cycle, we can see that PG-rules or PR-rules have to be selected if the belief matches with guard conditions. A goal-plan tree can therefore be defined like CASO where all possible paths to achieving a given goal may be depicted. Thus instead of the agent selecting and executing any plan, it can use the objective function to determine the path that it can follow which would optimize the objective function. Moreover, the look-ahead mechanism could also be
made parametric where the user can give the number of steps (PR-Rules in the goal-plan tree) to look ahead. This could be achieved by modifying the source code (as in case of Jason for CASO) in such a way that the user could specify the number of steps at run-time through messages. It is to be noted here that 3APL does not have any intentions as it pursues the plan that it has currently selected.

In this trivial example, it is obvious that juice available will be chosen. This example shows a 1-step look-ahead but we can have multiple-step look-ahead where there could be several PR-Rules which in turn could call other PR-Rules thereby creating a goal-plan tree.

### 7.3 2APL

Unlike 3APL, 2APL [20] is designed to implement multi-agent systems. 2APL includes a modification of 3APL programming constructs and adds new constructs like events, goals, and a variety of action types such as external actions, goal related actions, and communication action. 2APL provides two additional rule types to implement an agents reactive and proactive behavior. While 3APL rules can be applied to revise any arbitrary plan at run-time, the corresponding modified 2APL rules are applied only to revise plans when their executions are failed. 2APL proposes a new plan constructs to implement a non-interleaving execution of plans. In 3APL, at each cycle the agent’s plans are executed and revised, while in 2APL at each cycle goals, (internal and external) events, and messages are processed as well. In 2APL, the capabilities from 3APL have been renamed to belief updates and represent mental actions which are used to update the beliefbase.

One of the new features of 2APL is the procedural call rules. These rule control event and message handling. They prescribe a plan, which is added to the current
7.3. 2APL

plan whenever a given condition occurs. A message event is created for each message received and an event event for each occurred event, to allow PC-rules to fire on them. This provides a formal handling of events and messages, unlike the headless PG-rules and common PR-rules of 3APL. PC-rules can also be used to call imperative procedures through the use of abstract actions.

7.3.1 Deliberation Cycle of 2APL

The operations are segmented in three steps and is briefly explained below:

1. Events and Messages Processing, processes the existing events and incoming messages from the event base; the rules are defined by the procedure call rules (PC-Rules);

2. Plan Formation, processes the plan reasoning rules (PR-Rules), and;

3. Basic Action Execution, executes the physical and mentalist actions during plan execution.

7.3.2 Extending 2APL with constraint-based optimization

Like 3APL the semantics of 2APL do not specify the selection criterion when multiple plan rules are possible.

1. As with 3APL, it is not hard to see that the same logic of having beliefs as CLP and incorporating them into 2APL is possible. By incorporating beliefs as CLP, we relieve the programmer of specifying the rules for plan 2APL. Thus agent belief bases can be represented as CLPs instead of LPs (i.e., Prolog).
2. Optimisation objectives could be introduced which would be used for plan and intention selection. These objectives would come as non-triggering events and be stored in the objective store. CLPs are executed in the Plan Formation process where PG-rules are used to select the best plan based on the current beliefs and the objective store.

3. Parametric lookahead mechanism could be introduced where the PG-rules are decomposed into goal-plan tree.

4. Basic actions could have parameters that would be instantiated during action execution.

7.4 CANPLAN

CANPLAN [67] is based on BDI-style agent programming language and includes an on-demand planning mechanism in the style of Hierarchical Task Networks (HTN) [28]. CANPLAN, provides a flexible approach regarding when to perform full lookahead and is possibly more expressive than either BDI or HTN systems alone. CANPLAN is based on CAN [81] and AgentSpeak(L) [61]. Unlike AgentSpeak(L), the semantics for CAN includes both failure handling and declarative goals which are its main features. CANPLAN uses lookahead deliberation or hypothetical reasoning about the effects of one choice of expansion over another. This lookahead technique is used to guarantee goal achievability and to avoid undesired situations. An agent is created by the specification of a set of base beliefs and a set of plans. The belief base of an agent is a set of formulas from some (knowledge representation) logical language. Here actions are the usual basic means of the agent to change its environment may have preconditions and effects.
7.4. CANPLAN

7.4.1 Deliberation Cycle

CANPLAN provides a mechanism for local deliberation on-demand that the programmer can use, as opposed to a fixed integration of planning within the execution engine. By using the new local planning mechanism the programmer can rule out BDI executions that are bound to fail. An agent plan library $\prod$ consists of a collection of plan rules of the form $e : \Psi \leftarrow P$, where $e$ is an event and $\Psi$ is the context condition which must be true in order for the plan-body $P$ to be applicable. The plan-body $P$ is built from primitive actions $\text{act}$ that the agent can execute directly, operations to add $+b$ and delete $-b$ beliefs, tests for conditions $?\Psi$, and events or (internal) achievement goals !e. Complex plans can be specified using sequencing $P_1; P_2$, parallelism $P_1||P_2$, and declarative goals $\text{Goal}(\Psi_s, P, \Psi_f)$. A goal program $\text{Goal}(\Psi_s, P, \Psi_f)$ states that we should achieve the declarative goal $\Psi_s$ by using (procedural) program $P$; failing if $\Psi_f$ becomes true.

7.4.2 Extending CANPLAN with constraint-based optimization

It is to be noted that the lookahead of CANPLAN is used for failure conditions whereas the lookahead in CASO is for the best plan and intention selection. Moreover, the actions defined in CANPLAN may have preconditions and effects. The underlying basic infrastructure of CANPLAN is similar to that of AgentSpeak(L) and hence extensions to the language could be done in a similar manner as that done for CASO.

The extensions I propose in CANPLAN are:

1. Incorporation of parametric look-ahead mechanism for plan and intention selection. This look-ahead is different from the lookahead mechanism in CANPLAN
and one could possibly combine the two types of look-ahead to build a robust system.

2. **Extending the belief base to incorporate CLPs.** In CANPLAN, the belief base normally contains ground belief atoms in a first-order language and this can be extended easily without any loss of generality.

3. **Incorporation of objective function as non-triggering event.** Objective functions could be treated as a non-triggering event that updates an objective store which is used for selecting the first plan to execute.

4. **Addition of action parameters.** Plans could be modified so that besides preconditions and effects, one could also have parameters for actions which would be instantiated during intention selection.

The agents in CANPLAN respond to events. The interpreter would match the context of each plan with the external triggering event and this would generates a set of all relevant plans. The context part of the plans is united against the agents beliefs (which includes constraints). Constraint solving is now performed on these relevant plans to determine whether the constraint(s) in the context of the plan is (are) consistent with the belief base. This results in a set of applicable plans which can actually be used at that moment for handling the chosen event. Unlike AgentSpeak(L) there is no option selection function and in order to handle plan failures, all the applicable plans along with their context conditions are put together in the intention base. The order in which the plans are put together is now based on the look-ahead technique used in CASO where a goal-plan tree is generated. The agent objectives could be introduced as objective functions into the agent system as non-triggering events just as it had been done in CASO. The objectives would in turn be stored in an objective store which would be consulted when ordering the plans in the intention.
The best plan is put in front followed by the next best and so on. Hence, if the execution of the first plan fails for some reason, the next intention (which is similar to first one but now does not include the pair of the first plan and the context condition) will be attempted. There would be no change in the way the rest of the interpreter works in CANPLAN.
Chapter 8

Conclusion

In this chapter I present a summary of my thesis and also discusses on some of the future work that I plan to undertake.

8.1 Summary and Discussion

In this thesis I presented a method of combining BDI based programming with constraint logic programming (CLP) techniques to provide a single agent in a multi-agent system to equip itself with deliberative techniques in addition to its reactive properties. I also proposed a new language CASO, which provides the user with the flexibility of adding explicit objectives and constraints to achieve final goals. CASO uses a modified version of Jason, the well-known BDI AgentSpeak(L) interpreter, together with another open-source constraint solver ECLiPSe thereby combining reactive combining agent programming with constraint solving techniques. CASO is based on the strong theoretical foundations of BDI and the formal semantics as well as implementation details has been presented.

I also made further extensions to CASO to create a new language called BAOP. BAOP incorporates user preferences into the BDI framework. This work uses soft
constraints in an agent environment where I give a quantitative dimension to this agent deliberation process by apply c-semiring based techniques to determine the preferred solution.

I have taken a real world problem in the supply chain domain and show how CASO may be used in such a domain. This shows how CASO can be deployed in many agent application domains like supply chain, health care etc. as well as used in the design and simulation of such applications where several types of decision making and optimisations may be required. Finally, I demonstrated how the approach of incorporating CLP into AgentSpeak(L) can be applied to other BDI languages like 3APL, 2APL and CANPLAN. This shows the universal appeal of this technique.

While there has been some work in guiding an agent to select the best plan based on preferences, my approach is different from the existing ones mentioned earlier. Unlike most other work, I assume that the agent environment is highly reactive in nature and therefore constraints, preferences and objectives could change at every point in time. I give the user the ability to change objectives and preferences thereby making this a highly flexible system.

The publications resulted from work in this thesis are [16], [17], [18] and [19].

8.2 Future Work

In this thesis my work has been with respect to a single agent and where I described how an agent is able to deliberate on its actions and choose the best path to reach its goal. This work could be further extended in a multi-agent environment where each agent would have a set of preferences and there could be a global preference set that would try to achieve an overall system optimization across all agents. This is also
particular useful in agent negotiation where each agent tries to negotiate in order to achieve its own goal.

I have given example of biomass supply chain in this thesis and would like to extend this idea to other problem domains like health care and environment which could benefit from the optimizing capabilities of the BDI agent. I would like to develop good elicitation strategies that reduce the time to find out the preferred plan quickly. I plan to develop more efficient techniques for selecting the best plan in a complex environment which has a huge range of options to execute at any point in time. Another option is to incorporate case-based reasoning strategies into the BDI system. These could include storing the outcome of plan execution (cases) in a memory which could be used to guide the agent to reason about its future actions.

In the future, I plan to develop a more generic framework for BDI model whereby preferences, objectives, optimisations and look-ahead techniques could be embedded so that the agents could truly act as optimizers in real time complex environments.
Appendix

In this appendix I present the complete CASO agent code for the Biomass Supply Chain example as described in chapter 6.

CASO plans

Plan #1:
+!cut() : tractor(grappler) & machine($M_1$)← moveWholeToStorage(CostToStorage);
+atStorage(Whole); !transportWholeToPlant();

Plan #2:
+!cut(): tractor(forestry) & machine($M_2$)← moveStickToStorage(CostToStorage);
+atStorage(Stick); !transportStickToPlant();

Plan #3:
+!cut(): tractor(flatbed) & machine($M_2$)← moveStickToStorage(CostToStorage);
+atStorage(Stick); !transportStickToPlant();

Plan #4:
+!cut(): tractor(bulkCommodity) & machine($M_3$)←
moveChipToStorage(CostToStorage); +atStorage(Chip); !transportChipToPlant();

Plan #5:
+!cut(): tractor(forestry) & machine($M_4$)← moveBilletToStorage(CostToStorage);
8.2. Future Work

+atStorage(Billet);!transportChipToPlant();

Plan #6:
+!cut(): tractor(forestry) & machine(M_5) ← moveRndBaleToStorage(CostToStorage);
+atStorage(RndBale);!transportRndBaleToPlant();

Plan #7:
+!cut(): tractor(flatbed) & machine(M_5) ← moveRndBaleToStorage(CostToStorage);
+atStorage(RndBale);!transportRndBaleToPlant();

Plan #8:
+!cut(): tractor(forestry) & machine(M_6) ← moveSqBaleToStorage(CostToStorage);
+atStorage(SqBale);!transportSqBaleToPlant();

Plan #9:
+!cut(): tractor(flatbed) & machine(M_6) ← moveSqBaleToStorage(CostToStorage);
+atStorage(SqBale);!transportSqBaleToPlant();

Plan #10:
+!transportWholeToPlant() : truck(highSided) & atStorage(Whole) ←
loadTruck(T_{M_1}, T_{M_2}); moveToPlant().

Plan #11:
+!transportWholeToPlant() : truck(flatDeck) & atStorage(Whole) ←
loadTruck(T_{M_1}, T_{M_2}); moveToPlant().

Plan #12:
+!transportSticksToPlant() : truck(highSided) & atStorage(Sticks) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant().

Plan #13:
+!transportSticksToPlant() : truck(flatDeck) & atStorage(Sticks) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant().

Plan #14:
8.2. Future Work

+!transportChipToPlant() : truck(flatDeck) & bin() & atStorage(Chip) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**Plan #15:**
+!transportChipToPlant() : truck(highSided) & atStorage(Chip) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**Plan #16:**
+!transportBilletToPlant : truck(flatDeck) & bin() & atStorage(Billet) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**Plan #17:**
+!transportBilletToPlant : truck(highSided) & atStorage(Billet) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**Plan #18:**
+!transportRndBalesToPlant : truck(flatDeck) & atStorage(RndBales) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**Plan #19:**
+!transportSqBalesToPlant : truck(flatDeck) & atStorage(SqBales) ←
loadTruck(T_{M_1}, T_{M_2}); transportToPlant() .

**CASO beliefs**

**trailer(grappler):**

\[
\begin{align*}
\text{Vars} &= [\text{TotalWeight}, \text{CostPerHr}, \text{TripPerHr}, \text{CapacityPerTrip}], \\
\text{CostPerHour} &\#= 25, \\
\text{TotalWeight} &\#= 100, \\
\text{TripPerHour} &\#= < 4,
\end{align*}
\]
CapacityPerTrip #=< 5,
CostToStorage #= CostPerHr*TotalWeight/(TripPerHr * CapacityPerTrip)

\texttt{trailer(\textit{forestry})}:-
\begin{align*}
\text{Vars} &= [\text{TotalWeight, CostPerHr, TripPerHr, CapacityPerTrip}], \\
\text{CostPerHour} &= 22, \\
\text{TotalWeight} &= 100, \\
\text{MaxWorkingHours} &= 10, \\
\text{TripPerHour} &=< 6, \\
\text{CapacityPerTrip} &=< 8, \\
\text{CostToStorage} &= \text{CostPerHr*TotalWeight}/(\text{TripPerHr} \times \text{CapacityPerTrip})
\end{align*}

\texttt{trailer(\textit{flatbed})}:-
\begin{align*}
\text{Vars} &= [\text{TotalWeight, CostPerHr, TripPerHr, CapacityPerTrip}], \\
\text{CostPerHour} &= 21, \\
\text{TotalWeight} &= 100, \\
\text{TripPerHour} &=< 6, \\
\text{CapacityPerTrip} &=< 8, \\
\text{CostToStorage} &= \text{CostPerHr*TotalWeight}/(\text{TripPerHr} \times \text{CapacityPerTrip})
\end{align*}

\texttt{truck(\textit{highSided})}:-
\begin{align*}
\text{Vars} &= [\text{TotalWeight, DieselPrice, FuelPerTrip, TruckCapacityPerTrip, Trips}], \\
\text{TotalWeight} &= 100, \\
\text{DieselPrice} &= 1.6, \\
\text{FuelPerTrip} &= 30, \\
\text{TruckCapacityPerTrip} &=< 15,
\end{align*}
8.2. Future Work

\[ \text{Trips} \leftarrow \frac{\text{TotalWeight}}{\text{TruckCapacityPerTrip}}, \]
\[ \text{CostToPlant} \leftarrow \text{DieselPrice} \times \text{FuelPerTrip} \times \frac{\text{TotalWeight}}{\text{TruckCapacityPerTrip}}. \]

\text{truck(flatDeck):-}

\text{Vars} = [\text{TotalWeight}, \text{PetrolPrice}, \text{FuelPerTrip}, \text{TruckCapacityPerTrip}, \text{Trips}],

\text{TotalWeight} \leftarrow 100,
\text{PetrolPrice} \leftarrow 1.5,
\text{FuelPerTrip} \leftarrow 20,
\text{TruckCapacityPerTrip} \leftarrow 10,
\text{Trips} \leftarrow \frac{\text{TotalWeight}}{\text{TruckCapacityPerTrip}},
\text{CostToPlant} \leftarrow \text{PetrolPrice} \times \text{FuelPerTrip} \times \frac{\text{TotalWeight}}{\text{TruckCapacityPerTrip}}.
References


[38] Koen V. Hindriks, Frank S. de Boer, W.V. der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. In *Autonomous Agents and Multi-Agent Systems*,}


References


agents and multiagent systems (AAMAS ’07), pages 1–8, New York, NY, USA, 2007. ACM.


[73] C. R. Stucley, Rural Industries Research, Development Corporation (Australia), and Joint Venture Agroforestry Program (Australia). Biomass energy production in Australia : status, costs and opportunities for major technologies / by C.R. Stucley ... [et al.]. Rural Industries Research and Development Corporation, Canberra :, 2004.


[75] John Thangarajah, Lin Padhgam, and Michael Winikoff. Detecting and avoiding interference between goals in intelligent agents. In G. Gottlob and T. Walsh, edi-


