# Implementing reactive BDI agents with user-given constraints and objectives

Aniruddha Dasgupta
Decision Systems Lab
School of Computer Science and Software
Engineering
University of Wollongong
Wollongong, NSW 2522, Australia
ad844@uow.edu.au

Aditya K.Ghose
Decision Systems Lab
School of Computer Science and Software
Engineering
University of Wollongong
Wollongong, NSW 2522, Australia
aditya@uow.edu.au

## ABSTRACT

CASO is an agent-oriented programming language based on AgentSpeak(L), one of the most influential abstract languages based on the BDI (Beliefs-Desires-Intentions) architecture. 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. CASO incorporates constraints and objectives into the symbolic approach of reactive BDI model which lead to better expressive capabilities as well as more efficient computation. Jason is a fully-fledged interpreter for a much improved version of AgentSpeak(L). In this work we modify Jason to incorporate the operational semantics of CASO. CASO also uses ECLiPSe, an open source constraint solver, to apply constraint solving techniques. Our preliminary results show that CASO can be used as a powerful multi agent programming language in solving problems in complex application domains.

## 1. INTRODUCTION

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. One of the most popular and successful framework for Agent technology is that of Rao and Georgeff [11], in which the notions of *Belief*, *Desire* and *Intention* or *BDI* are central. Beliefs represent the agent's 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. Desires represent a state which the agent is trying to achieve. Intentions are the chosen means to achieve the agent's desires, and are generally implemented as plans and

post-conditions. 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. Besides these components, the BDI model includes a plan library, namely a set of "recipes" representing the procedural knowledge of the agent, and 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. Usually, BDI-style agents do no adopt 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. The BDI model provides that all the knowledge of a rational agent about the world is organized in statements that are its beliefs. An agent's desires depict some states of the world that the agent "would like" to be realized. 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.

In this paper present an implementation our design of CASO agent [7], discussing in more detail: (a) the requirements specification with respect to CASO; (b) an implementation of the CASO design; (c) some example exploring the CASO design and and finally (d) the strengths and weaknesses of our design. From an implementation point of view the existence of special libraries or dedicated programming languages that provide data and control structures for manipulating agent specific properties allows for an easy implementation of agent models. The remainder of this article is organized as follows. Section 2 gives a brief introduction of popular BDI language AgentSpeak(L) [10] as well as talks briefly about Jason [5], an interpreter of AgentSpeak(L); section 3 discusses ECLiPSe [2], a constraint programming toolkit respectively; section 4 describes the language CASO and gives an overview of its operational semantics; section 5 describes the implementation details; section 6 gives an example of using CASO and section 7 provides some experimental results. Finally, the last section describes some related work and gives concluding remarks.

## 2. AGENTSPEAK(L)

AgentSpeak(L) is an agent framework/language with explicit representations of beliefs and intentions for agents. This agent programming language was initially introduced by Rao [10]. 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 not explicitly represented as modal formulas. Instead, designers can ascribe these notions to agents written in AgentSpeak(L). 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. AgentSpeak(L) agent is described as a set of $\langle E, B, P, I, A, S_E, S_O, S_I \rangle$ 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 $P$.

- $S_I$ 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). There are two types of goals in AgentSpeak(L). An "achievement goal" (a predicate prefixed with "!"), states that the agent wishes to achieve a state of the world in which the associated predicate is true. A "test goal" (a predicate prefixed with "?"), states that the agent wishes to test if the associated predicate is a true. 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. An agent must have pre-designed plans in its plan library to handle the incoming internal or external events. 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.

One of the most popular fully-fledged interpreter of AgentSpeak(L) is **Jason**. 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-customisable features. Jason is implemented in Java (thus multi-platform) and is available Open Source, distributed under GNU LGPL.

## 3. ECLIPSE: A CONSTRAINT SOLVER

Constraint satisfaction is a powerful computational paradigm which proposes techniques to find assignments for problem variables subject to constraints on which only certain combinations of values are acceptable. The success and the increasing application of this paradigm in various domains mainly derive by the fact that many combinatorial problems can be expressed in a natural way as a Constraint Satisfaction Problem (CSP), and can subsequently be solved by applying powerful CSP techniques.
ECLiPSe 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 modelling, constraint programming, mathematical programming, and search techniques. It contains several constraint solver libraries, a high-level modelling and control language, interfaces to third-party solvers, an integrated development environment and interfaces for embedding into host environments.

## 4. CASO: A REACTIVE BDI LANGUAGE

The concept of using constraints and explicit objectives in a high-level agent specification language like Agentspeak(L), yields significant advantages in terms of both expressivity and efficiency as shown in our previous work in [8]. The improvised technique applies constraint and objective directed solving on the context section of a BDI agent's plan specification in order to determine an application plan to fire. CASO (**C**onstraint **A**gent**S**peak(L) with **O**bjective) is a programming language based on the popular BDI language AgentSpeak(L)which incorporates constraints and objectives into the symbolic approach of BDI model. CASO incorporates Constraint Solving and Optimization (CSOP) techniques where the optimization is based on the objective function (softgoal).
In CASO, one can express agents' goals quantitatively - for example, agents can have some utility (objective) function which needs to be maximized. Incorporating constraints into a reactive BDI agent programming language can lead to better expressive capabilities as well as more efficient computation (in some instances). 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. CASO also incorporates efficient option 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. The typical CASO execution cycle is characterized by the following steps:

1. observe the world and the agent's internal state, and update the event queue consequently;

2. generate possible new plan instances whose trigger event matches an event in the event queue (relevant plan instances) and whose precondition (beliefs and constraints in plan body) is satisfied (applicable plan instances); plan selection is based on the satisfiability of the current set of constraints as well as the one which maximizes the current objective(using look-ahead techniques);

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, 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.

Informally, an agent program in CASO consists of a set of beliefs B, a set of constraints C, an objective function O, a set of events E, a set of intention I, a plan library P, a constraint store CS, an objective store OS and three selection functions $S_E, S_P, S_I$ to select an event , a plan and an intention respectively to process and $n_p$ and $n_i$ are the two parameters which denote the number of steps to look-ahead for plan and intentions selection respectively.

**Definition**: *A CASO agent program is a tuple* $\{B, P, E, I, C, O, S_O, S_E, S_I, n_p, n_i, CS, OS\}$ *where*

- *B is a set of Beliefs.*

- *P is agent plan repository, a library of agent plans.*

- *E is set of events (including external and internal).*

- *I is a set of intentions.*

- *C is a set of constraints.*

- *O is an objective function.*

- *$S_E$ 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*

- *CS is a constraint store which stores constraints which come as events.*

- *OS is an objective store which stores the objective function which comes as an event.*

- *$n_p$ is an integer which denotes the number of steps required to look-ahead for plan selection.*

- *$n_i$ is an integer which denotes the number of steps required to look-ahead for intention selection.*

In CASO, a constraint directed improvisation is incorporated into the computation strategy employed during the interpretation process. Constraint logic programming (CLP) combines the flexibility of logic with the power of search to provide high-level constructs for solving computationally hard problems such as resource allocation.
*Formally, a language CLP(X) is defined by*

- *constraint domain X,*

- *solver for the constraint domain X and*

- *simplifier for the constraint domain X*

*A CASO **plan** p is of the form* $t : b_1 \wedge b_2 \wedge \cdots \wedge b_n \wedge c_1 \wedge c_2 \wedge \cdots \wedge c_m \leftarrow sg_1, sg_2, \cdots, sg_k$ *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 we will use *BContext(p)* to denote the belief context of plan p. Thus

$$BContext(p) \equiv b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

Similarly, we will use *CContext(p)* to denote the constraint context of plan p. Thus

$$CContext(p) \equiv c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

Transition of agent program to process events depends on the event triggers. An event trigger, $t$, can be addition(+) or removal(-) of an achievement goal($\pm !g_i$) or a belief($\pm b_i$).

## 4.1 Informal Semantics

The CASO interpreter manages set of events, a constraint store, a objective store and a set of intentions with three selection functions. Intentions are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. Events, which may start off the execution of plans that have relevant triggering events, can be external when originating from perception of the agentš environment (i.e., addition and deletion of beliefs based on perception are external events) ; or internal, when generated form the agentš own execution of a plan (i.e., as subgoal in a plan generates an event of the type addition of an achievement goal).
In the latter case, the event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed on top of that intention). External events create new intentions, representing separated focuses of attention for the agentš acting on the environment.
The constraint store is initialized by the relevant constraints whenever a trigger contains a constraint in its context. At every cycle of the interpreter, the constraint store is enhanced with new constraints when applicable selected plan is executed. These incremental constraints collecting process eventually leads to a final consistent constraints set. Constraint solving is applied to the context of each plan to determine applicable plans as well as to generate solutions for subsequent actions. Similarly, the objective store contains the set of objective functions that need to be maximized (or minimized) which are part of the event context and is similarly updated at each cycle. *Plan Selection* is described in detail in the next subsection.
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, and the formula in the beginning of its body is taken 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 selected formula is 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 (the interpreter informs to the architecture component responsible for the agent effectors what action is required). 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.

## 4.2 Plan selection with parametric look-ahead

After $S_E$ has selected an event, CASO has to unify that event with triggering events in the heads of plans. This generates a set of all *relevant plans*. The constraints (if any) that are included in the constraint part of the context are put in the *constraint store*. The context part of the plans is unified against the agents beliefs. 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 constraints already collected in the constraint store . This results in a set of *applicable plans*(plans that can actually be used at that moment for handling the chosen event).
The objective store maintains a set of objective function which may be present in the event context. At each interpreter cycle, the objective store is also updated with an *objective function* for maximizing (or minimizing).

Given plans $p_1$ and $p_2$ in the plan library, and given a current constraint store $C$ and a current objective store $O$, $p_1 \leq_{opt} p_2$ if and only if: $OptSol(C \cup CContext(p_1), OS) \geq OptSol(C \cup CContext(p_2), O)$ .

$OptSol(Constraints, Objective)$ denotes the value of the objective function when applied to the optimal solution to the problem denoted by the pair (Constraints, Objective). We assume of course that $C \cup CContext(p_1)$ and $C \cup CContext(p_2)$ are solvable.
Optimization techniques are now applied by the optimizer to each of the applicable plan to determine an optimal solution. In effect we are solving a 'Constraint Satisfaction Optimisation Problem' (CSOP) which consists of a standard 'Constraint Satisfaction Problem' (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, which becomes the intended means for handling that event, and either pushes that plan on the top of an existing intention (if the event was an internal one), or creates a new intention in the set of intentions (if the event was external, i.e., generated from perception of the environment). 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. Plan selection is defined as follows:

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 $p_i \in AppPlans(t)$.

The agent program is also responsible for making sure that the objective store is consistent at any point of time. During each cycle of the interpreter, new objectives are added into the objective store and hence a consistency checker is used to maintain consistency. Formally a *consistent objective store* is defined as below.

Given an objective store $OS$ and a new objective $f$, the result of augmenting $OS$ with $f$, denoted by $OS_f^*$, is defined as $\gamma(MaxCons(OS \cup f))$ where $\gamma$ is a choice function and $MaxCons(X)$ is the set of all $x \subseteq X$ such that

1. $x$ is consistent and

2. there exists no $x'$ such that $x \subset x' \subseteq X$ and $x'$ is consistent

The new objective store is now given by $\gamma(MaxCons(OS \cup \overline{O}) \cap OS)$ where $\gamma$ is the choice function, $OS$ is the objective store and $\overline{O}$ is the negation of the objective $O$.
Selection of O-preferred plan can be further enhanced by using $n_p$ the look-ahead parameter form plan selection. In case $n_p$=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 $n_p > 0$ then a look-ahead algorithm (used for choosing the next move in a two-player game) is performed to select the O-preferred plan.
We assume that the agent is trying maximize its objective function and the environment may change in the worst possible way which would minimize the objective function. The goal of the agent would be to select a plan which would maximize the minimum value of the objective function resulting from the selection of plans which may occur due to the set of new possible events that may come from the environment.
We follow the definition of *goal-plan tree* given in [13] to decompose the set of plans into a tree structure. In CASO, goals are achieved by executing plans and each goal has at least one plan, if not many, 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).
Each goal-plan tree consists of - a number of 'AND' nodes which are subgoals that must be executed sequentially for the goal to succeed; and a number of 'OR' nodes which are subgoals any one of which must be executed for the goal to succeed. Given a set of applicable plans, an agent would always try to achieve this objective at every decision step. However, there could be unforeseen situations which may result in the agent changing its normal course of action at any of these decision points. Thus the strategy for the agent is to compute in advance the worst case scenario that may occur due to the change in the highly dynamic environment. Figure 1 shows the tree decomposition for plan P depicting all possible choices (OR nodes). The numbers corresponding to the leaf nodes are the values of the optimization function (say, $f$) which we are trying to maximize. Using the *LookAheadPlanSelection* look-ahead algorithm shown in Algorithm 1, we obtain the value of 3 at the root node which suggest that the agent should follow plan P2.
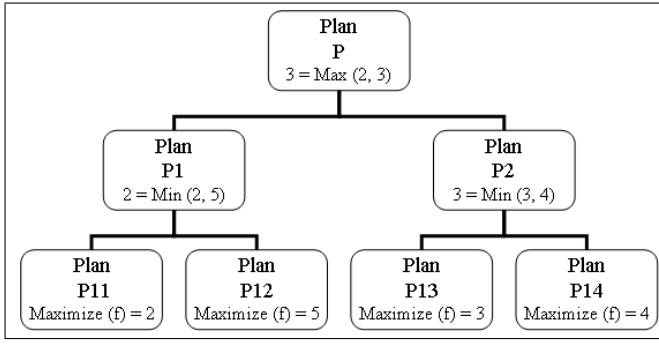
**Figure 1: Plan Tree**
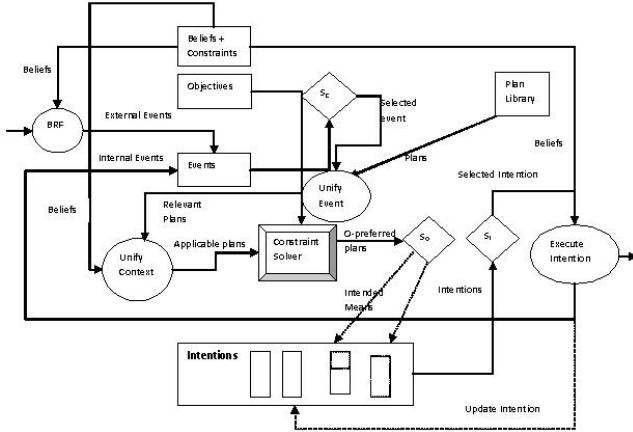
## 5. IMPLEMENTING CASO



**Figure 2: Operational Semantics of CASO**

In this section we give some more details on the implementation of a CASO interpreter, which is clearly depicted in Figure 2 (modified from [5]). The pictorial description of such interpreter, greatly facilitates the understanding of the interpreter. In the figure, sets (of beliefs, events, plans, and intentions) are represented as rectangles. Diamonds represent selection (of one element from a set). Circles represent some of the processing involved in the interpretation of CASO programs. The 3-d box represents the ECLiPSe CLP solver that is plugged into the system which is responsible for the option selection function based on the set of objectives and beliefs and/or constraints.

### 5.1 ECLiPSe plug-in for option selection function

As mentioned in earlier sections, we use ECLiPSe for option 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. Beliefs in CASO are written as ECLiPSe CLP programs

---

**Algorithm 1** LookAheadPlanSelection(int n, state S, ObjectiveStore OS, ConstraintStore CS)

1: Generate *goal-plan tree* up to $n$ levels from current state $S$ comprising of subgoals of AND and OR nodes with subplans.
2: Start from the root node.
3: Let constraint store at node $p = c_p$.
4: Let $o_p$ denote the value of objective function at node $p$.
5: For each node $p$ in the goal plan tree set $c_p \leftarrow CS$
6: **if** node p has child nodes $p1, p2 \cdots, pk$ in an AND structure **then**
7: 　Apply constraint solving at each $pi$ with the current constraint store $c_{pi}$ and the set of constraints for $pi$ to obtain $o_{pi}$.
8: 　Set $c_{pi+1} \leftarrow c_{pi}$ for all $i \geq 1$.
9: 　Initialize constraint store for all child nodes of each $p_i$ with $c_{pi}$.
10: **end if**
11: **if** node p has child nodes $p1, p2 \cdots, pk$ in an OR structure **then**
12: 　Compute the objective function and update the constraint store for each $pi$.
13: 　Initialize constraint store for all child nodes of each $p_i$ with $c_{pi}$.
14: **end if**
15: **while** $n \neq 1$ **do**
16: 　Propagate minimum value of objective function up to each parent node starting from the leaf node.
17: 　$n = n - 1$.
18: **end while**
19: Propagate the maximum value of its children for state $S$.
20: At state $S$, the best plan is the child with the maximum value.

---

as shown in figure 3 below where *Vars* represent the set of variables. Constraints are defined on the variables as shown below. In the example below, let us assume that the belief *resource_available()* is a part of the agent belief base. As per CASO, this belief is stored as in a file *resource_available.ecl*. In this example, when the predicate *resource_available()* is executed as a query, the solver would solve the optimization as per the objective shown in the figure and generate a solution. The solver first calls the *eplex* (External CPLEX Solver Interface) library which allows an external Mathematical Programming (MP) solver to be used by ECLiPSe. *eplex* is one of the most widely distributed, scaleable and efficient packages incorporating a linear constraint solver. A problem in ECLiPSe is modeled by a set of simultaneous equations: 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. The *eplex* library allows for the user to write programs that combines MP's global algorithmic solving techniques with the local propagation techniques of Constraint Logic Programming. The result of the objective function along with the instantiated variables (Vars) is stored in an output file *resource_available.txt*. A CASO agent may not have any objective function initially in which case there would be no function to minimize and the solver may choose to instantiate the variables with a possible set of variable instantiation based on the set of constraints. In

case the objective comes externally from the environment, the file *resource_available.ecl* is modified and specific objective function is written into the file. When the objective changes, i.e. a new objective comes to the objective store, this file is re-written with the new objective function. 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 which produces the highest value of the objective function and is then pushed as an intention. In case there are actions associated with plan whose parameters are part of the ECLiPSe CLP, the variable values are also pushed along with the intention.

---

*Belief*:
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).

*Optimization function:*
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))

---

**Figure 3: CASO Belief and Objective function written in ECLiPSe CLP style**

## 5.2 Modifying Jason

We are not going into the details of our modifications but we only describe the important changes to Jason in this section. Since we have kept the essence of Jason interpreter, the only notable change we did has been with regards to the new operational semantics that has been described earlier. In particular, our main modifications to Jason are the following:

- CLP-style beliefs (with constraints and objectives) are now written in a separate file that can be modified by an external event when a new objective is added or deleted. The ECLiPSe solver 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.

Any application that can be deployed in Jason can also be deployed in CASO with the added benefit of application of an objective function that can be user defined which can change with every interpretation of CASO execution cycle by an external event.

## 6. EXAMPLE: USING CASO IN REAL-TIME DECISION MAKING FOR BIOMASS SUPPLY CHAIN

Our implementation of CASO as described earlier, can be shown by the following example where we try to describe the benefit of using CLP in agent paradigm. The example is chosen from a supply chain system where we describe the optimizations carried out by a single agent.

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 which is described in detail in [1]. The activities can include ground preparation and planting, cultivation, harvesting, handling, storage, in-field/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. Also, given the typical locations for biomass fuel sources (i.e. on farms and in forests) the transport infrastructure is usually such that road transport will be the only potential mode for collection of the fuel. In order to supply biomass from its point of production to a power station the following activities are necessary:

- Harvesting of the biomass in the field/forest.

- In-field/forest handling and transport to move the biomass to a point where road transport vehicles can be used

- Storage of the biomass. Many types of biomass will be harvested at a specific time of year 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 farm/forest, at the power station or at an intermediate site.

- Loading and unloading road transport vehicles. Once the biomass has been moved to the roadside it will need to be transferred to road transport vehicles for conveyance to the power station. At the power station the biomass will need to be unloaded from the vehicles.

- 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.

- Processing of the biomass to improve its handling efficiency and the quantity that can be transported. This can involve increasing the bulk density of the biomass (e.g. processing forest fuel or coppice stems into wood chips) or unitising the biomass (e.g. processing straw or miscanthus in the swath into bales). Processing can occur at any stage in the supply chain but will often precede road transport and is generally cheapest when integrated with the harvesting

Figure 4 describes the various options faced in the Biomass SCM. The figure shows that there are several decision points in the SCM which affect the final outcome. As an example, trees grown on farmland on a short rotation coppice basis can either be harvested as five meter whole sticks or cut and immediately processed into wood chips. Different harvesting
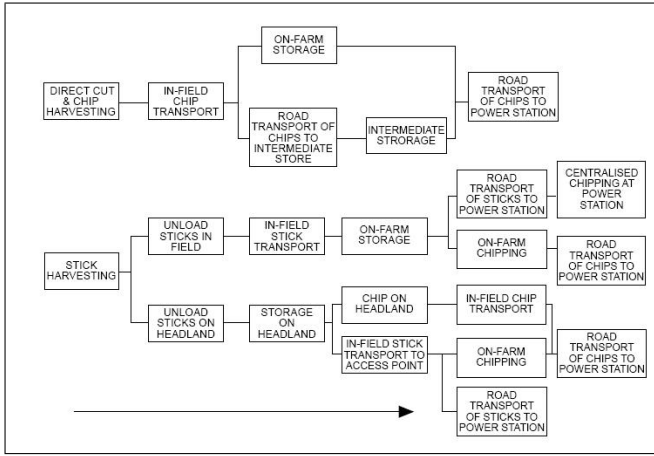
**Figure 4: Decision making in Biomass Supply Chain**

machinery is required for each of these harvesting systems. While sticks can be stored on the edge of fields without experiencing decomposition, chips need to be stored on at least a hard standing, and in a covered environment if decomposition is to be prevented. Therefore, the storage requirements for the two products are very different. Similarly, sticks and chips require very different transport systems in terms of both handling (i.e. loading and unloading vehicles) and suitable transport vehicle bodies for carrying them.. The easiest and most cost effective harvesting system can result in the need for expensive storage systems and can even lead to an inability to supply fuel of the desired quality to the power station.

The SCM as shown in the figure can be represented by a CASO agent program (Figure 5) which has several plans at its disposal and can choose one of the possible alternate plans based on the current set of constraints as well as the current objective that the SCM is trying to optimize. If we look into Plan#1, we can see that one of the subgoals of stick harvesting is to either chip on field or transport the sticks as denoted by *fieldChipOrTransport()*. These two subgoals are further elaborated in plans Plan#2 and Plan#3 where two possible course of action could take place depending on whether transport to power station is available or chipping on farm is possible. These two possible options are denoted by *transportAvailableToPowerStationFromField()* and *chippingPossibleOnFarm()* as two beliefs which can be written in ECLiPSe style CLP and is shown in Figure 6. For the sake of simplicity, let us assume that the objective function (minimize cost) is given by:

*optimize(min(50\*StickDeliveryTrucksReqd +*
*60\*LoadingRobotReqd + 75\*StickAsseblersReqd + 50\*ChipDeliveryTrucksReqd + 70\*ChippingMachineReqd +*
*65\*ChipLoadersReqd)).*

The numbers denote the *$ cost* and the variables (shown in figure 6) denote the quantity of each resource required. The first 3 variables are related to stick transport and the last 3 are related to chipping on field. If we look carefully into the two beliefs, we see that when the agent tries to select either of plans Plan#2 and Plan#3 in figure 5, it evaluates the context of each plan as given by beliefs Belief#1 and Belief#2 in figure 6, and finds out that both plans can equally be selected as all constraints are satisfied. However, once the
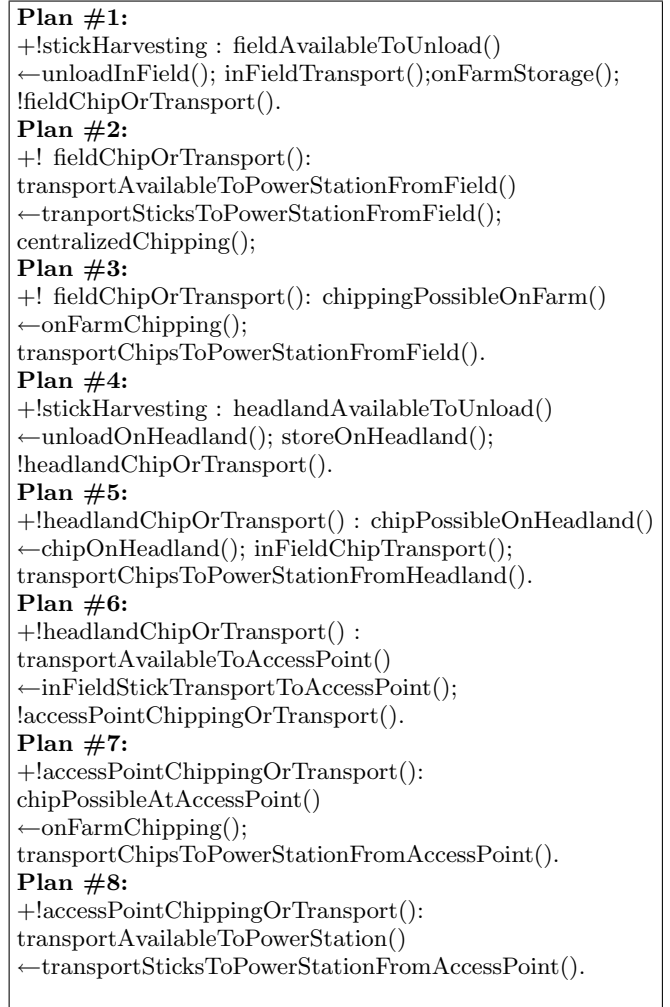
**Plan #1:**
+!stickHarvesting : fieldAvailableToUnload()
←unloadInField(); inFieldTransport();onFarmStorage();
!fieldChipOrTransport().
**Plan #2:**
+! fieldChipOrTransport():
transportAvailableToPowerStationFromField()
←tranportSticksToPowerStationFromField();
centralizedChipping();
**Plan #3:**
+! fieldChipOrTransport(): chippingPossibleOnFarm()
←onFarmChipping();
transportChipsToPowerStationFromField().
**Plan #4:**
+!stickHarvesting : headlandAvailableToUnload()
←unloadOnHeadland(); storeOnHeadland();
!headlandChipOrTransport().
**Plan #5:**
+!headlandChipOrTransport() : chipPossibleOnHeadland()
←chipOnHeadland(); inFieldChipTransport();
transportChipsToPowerStationFromHeadland().
**Plan #6:**
+!headlandChipOrTransport() :
transportAvailableToAccessPoint()
←inFieldStickTransportToAccessPoint();
!accessPointChippingOrTransport().
**Plan #7:**
+!accessPointChippingOrTransport():
chipPossibleAtAccessPoint()
←onFarmChipping();
transportChipsToPowerStationFromAccessPoint().
**Plan #8:**
+!accessPointChippingOrTransport():
transportAvailableToPowerStation()
←transportSticksToPowerStationFromAccessPoint().

**Figure 5: Plans related to a CASO Stick Harvesting Agent**

objective function is taken into consideration, the solution obtained from evaluating Belief#1 gives a value of 245 of the objective function and that from Belief#2 gives a value of 185. Based on the current scenario, the agent would choose Plan#1 as it gives lower cost. Tables 1 and 2 show the value of the objective function together with the value of the variables. Now, if the objective function changes at any point, the result obtained may be different and the agent would then choose a different plan based on the circumstances. As an example, the current objective is a function of *cost* but it can equally be made into a function of *time*. Thus, if the consideration is to minimize the amount of *time* that is required to carry out either of the two plans without any regards to the *cost*, a new objective function (minimize) could be written which is a function of *time*. The value of the variables are passed to the intention stack in the CASO interpreter cycle, and are used to initialize the parameters as described earlier. Also, the above example showed only 1-step look-ahead - however, one can easily go to reasonable desired depth of the decision tree (AND/OR goal-plan tree), and obtain all possible values of the objective function. The agent would then choose the plan that would be the best out

## BELIEF #1
**transportAvailableToPowerStationFromField()**:-
*% integer variables*
Vars = [StickDeliveryTrucksReqd, LoadingRobotReqd, StickAsseblersReqd],
integers(Vars),

*%constants*
StickDeliveryTrucksAvail=2,
LoadingRobotAvail =2,
StickAsseblersAvail =2,

*%inequality constraints*
*%enough trucks for transportation*
StickDeliveryTrucksReqd <=StickDeliveryTrucksAvail,

*%enough robots for loading sticks onto truck*
LoadingRobotReqd <=LoadingRobotAvail,

*%enough persons to assemble the sticks*
StickAsseblersReqd <=StickAsseblersAvail,

*%at least 3 robots + stick assemblers are required*
LoadingRobotReqd + StickAsseblersReqd >=3,

*%at least 1 delivery truck is required*
StickDeliveryTrucksReqd >=1,

*%equality constraints*:*total number of resources required*
StickDeliveryTrucksReqd + LoadingRobotReqd + StickAsseblersReqd =4

## BELIEF #2
**chippingPossibleOnFarm()**:-
*%integer variables*
Vars = [ChipDeliveryTrucksReqd, ChippingMachineReqd, ChipLoadersReqd],
integers(Vars),

*%constants*
ChipDeliveryTrucksAvail =2,
ChippingMachineAvail =2,
ChipLoadersAvail =2,

*%inequality constraints*
*%enough trucks for transportation*
ChipDeliveryTrucksReqd <=ChipDeliveryTrucksAvail,
*%enough chipping machines*
ChippingMachineReqd <=ChippingMachineAvail,
*%enough persons to load the sticks into chipping machine*
ChipLoadersReqd <=ChipLoadersAvail,
*%at least 1 machine, 1 loader and 1 delivery truck are required*
ChippingMachineReqd => 1,
ChipLoadersReqd >=1,
ChipDeliveryTrucksReqd >=1,

*%equality constraints* :*total number of resources required*
ChipDeliveryTrucksReqd + ChippingMachineReqd + ChipLoadersReqd =4

**Figure 6: Partial Set of beliefs related to a CASO Stick Harvesting Agent**

| Belief#1 | |
|---|---|
| StickDeliveryTrucksReqd | =1 |
| LoadingRobotReqd | =2 |
| StickAsseblersReqd | =1 |
| Objective | =245 |

**Table 1: Value of variables and objective function for Belief#1**

| Belief#2 | |
|---|---|
| ChipDeliveryTrucksReqd | =1 |
| ChippingMachineReqd | =1 |
| ChipLoadersReqd | =1 |
| Objective | =185 |

**Table 2: Value of variables and objective function for Belief #2**

of all possible worst cases as described in earlier section. It should also be noted here that the we are currently depicting only one agent in a multi-agent scenario which is doing decision making. However, this can easily be extended to a fully fledged MAS where several agents interact with each other and take their own decisions for optimizing their own objectives.

## 7. EXPERIMENTAL RESULTS

We ran a series of experiments to find out how the *quickly* the system could find the optimal plan. Our 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.
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. Note that the list of variables in the program has to be globally defined.

We 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 subplans, we calculated the time taken to find the optimum plan among the choices by fixing the branching factor, number of constraints per plan and the number of variables for a given objective function and varying the *depth* of look-ahead. We set *branching factor = 3, number of constraints/plan = 2* and *number of variables =5.*

**Experiment 2** Given a plan in a CASO program with multiple subplans, we calculated the time taken to find the optimum plan among the choices by fixing the look-ahead depth, the branching factor, number of constraints per plan variables and varying the number of *variables*. Note that for this experiment, we 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, we used similar objective function with lesser variables. We set *branching factor = 3, number of constraints/plan = 2* and *look-ahead depth =3*.

**Experiment 3** Given a plan in a CASO program with multiple subplans, we calculated the time taken to find the optimum plan among the choices by fixing the look-ahead depth, the branching factor, number of variables and the same objective function and varying the number of *constraints per plan*. Note that for this experiment, we 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). We set *branching factor = 3, look-ahead depth = 3* and *number of variables =5*.

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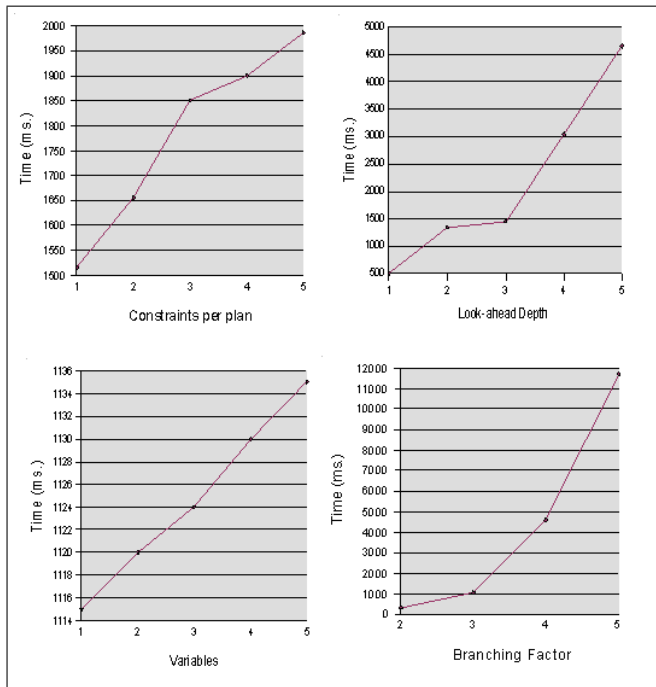


**Figure 7: Graphs showing experimental results**

Figure 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, we are 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, we 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 (Figure 7).

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 our implementation of CASO.

## 8. RELATED WORK AND CONCLUSION

Decision agents can be designed to provide interactive decision aids for end-users by eliciting their preferences and then recommending matching products. In [6] 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 language. AgentSpeak(XL) programming language [4] 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 [9]. Another related work on detecting and resolving conflicts between plans in BDI agents is presented in [14]. The "degree of boldness" of an agent is defined in [12] which represents he 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.

Our implementation of CASO 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 in the simple example described in earlier section, we can see that CASO can indeed 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 optimizations may be required. Moreover, the time taken to select a particular plan in real-time is very small (with a reasonable look-ahead depth) and is only

depended on the constraint-solver that we use. In future we plan to extend CASO to incorporate user preferences as c-semiring [3] and implement the design to create a more robust and powerful MAS which can be deployed in complex applications.

## 9. REFERENCES

[1] J. Allen, M. Browne, A. Hunter, J. Boyd, and H. Palmer. *Logistics management and costs of biomass fuel supply.* MCB UP Ltd., 1998.

[2] K. R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse.* Cambridge University Press, 2007.

[3] S. Bistarelli, U. Montanary, and F. Rossi. Semiring-based constraint satisfaction and optimisation. In *Journal of ACM.* ACM Press, 1997.

[4] R. Bordini, A. Bazzan, R. Jannone, D. Basso, R. Vicari, and V. Lesser. *AgentSpeak(XL):Efficient intention selection in BDI agents via decision-theoretic task scheduling.* ACM Press, 2002.

[5] R. Bordini, J. Hubner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, Ltd, 2007.

[6] S. Chalmers and P. M. D. Gray. Bdi agents and constraint logic. In *AISB Journal Special Issue on Agent Technology*, 2001.

[7] A. Dasgupta and A. K. Ghose. Dealing with objectives in a constraint-based extension to agentspeak(l). In *Proc. of the Pacific Rim International Workshop on Multi-Agents*, 2005.

[8] A. Dasgupta and A. K. Ghose. Caso: A framework for dealing with objectives in a constraint-based extension to agentspeak(l). In *Proc. of the 2006 Australasian Computer Science Conference*, 2006.

[9] J. Horty and M. Pollack. Evaluating new options in the context of existing plans. In *Artifical Intelligence*, 2001.

[10] A. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away: Proceedings of the 7th European WS on Modelling Autonomous Agents in a Multi-Agent World.* Springer-Verlag: Heidelberg,Germany, 1996.

[11] A. Rao and M. Georgeff. *BDI Agents: from theory to practice.* San Fransisco, USA, 1995.

[12] M. Schut and M. Wooldridge. Intention reconsideration in complex environments. In *Proceedings of International Conference on Autonomous Agents*, Varcelona, Spain, 2000.

[13] J. Thangarajah. Managing the concurrent execution of goals in intelligent agents. In *Phd. Thesis.* RMIT, 2004.

[14] J. Thangarajah, L. Padhgam, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *G. Gottlob and T. Walsh, editors, Proceedings of the International Joint Conference on Artificial Intelligence.* Academic Press, 2003.