

# Foundations of Flexible Multi-Agent Programming<sup>\*</sup>

Louise Dennis, Michael Fisher, and Anthony Hepple

Department of Computer Science, University of Liverpool, Liverpool, U.K.  
{L.A.Dennis,M.Fisher,A.J.Hepple}@csc.liv.ac.uk

**Abstract.** In this paper we are concerned with proposing, analyzing and implementing simple, yet flexible, constructs for multi-agent programming. In particular, we wish to extend programming languages based on the BDI style of logical agent model with two such constructs, *constraints* and *content/context sets*. These two aspects provide sufficient expressive power to allow us to represent, simply and with semantic clarity, a wide range of organisational structures for multi-agent systems. In summary, we not only motivate this approach, but provide its semantics, through modification of a simple semantics based on the core of AGENTSPEAK, 3APL and METATEM. In addition, we provide illustrative examples by simulating both constraints and content/context sets within the *Jason* interpreter for AGENTSPEAK. In summary, we advocate the use of these simple constructs in many logic-based BDI languages, by appealing to their expressive power, semantics and simple implementation.

## 1 Introduction

We can simply characterise an agent as an autonomous software component having certain goals and being able to communicate with other agents in order to accomplish these goals [15]. The ability of agents to act independently, to react to unexpected situations and to cooperate with other agents has made them a popular choice for developing software in a number of areas. At one extreme there are agents that are used to search the INTERNET, navigating autonomously in order to retrieve information; these are relatively lightweight agents, with few goals but significant domain-specific knowledge. At the other end of the spectrum, there are agents developed for independent process control in unpredictable environments. This second form of agent is often constructed using complex software architectures, and have been applied in areas such as real-time process control [12, 9]. Perhaps the most impressive use of such agents is as part of the real-time fault monitoring and diagnosis carried out in the NASA Deep Space One mission [10].

The key reason why an agent-based approach is advantageous in the modelling and programming of autonomous systems is that it permits the clear and concise representation, not just of *what* the autonomous components within the system do, but *why* they do it. This allows us to abstract away from the low-level control aspects and to concentrate on the key feature of autonomy, namely the goals the component has and the choices it makes. Thus, in modelling a system in terms of agents, we often describe each agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. Such agents then make decisions about what action to perform, given their beliefs and goals/intentions. This kind of approach has been

---

<sup>\*</sup> Work partially supported by EPSRC under grant EP/D052548.

popularised through the influential BDI (Belief-Desire-Intention) model of agent-based systems [12]. This representation of behaviour using *mental* notions is initially unusual, yet has several benefits. The first is that, ideally, it abstracts away from low-level issues: we simply present some goal that we wish to be achieved, and we expect it to act as an agent would given such a goal. Secondly, because we are used to understanding and predicting the behaviour of rational agents, the behaviour of autonomous software should be relatively easy for humans to understand and predict too. Not surprisingly, the modelling of complex systems, even space exploration systems, in terms of rational agents has been very successful [14, 13, 6]. Thus, the BDI approach to agent modelling has been successful. Unsurprisingly, this has led to many novel (usually, logic-based) programming languages based (at least in some part) upon this model; these are often termed *BDI Languages*. Although a wide variety of such languages have been developed [1] few have strong and flexible mechanisms for *organising* multiple agents, and those that do provide no agreement on their organisational mechanisms. Thus, while BDI languages have converged to a common core relating to the activity of individual agents [4], no such convergence is apparent in terms of multi-agent structuring.

In this paper we consider extending basic BDI languages with simple, yet powerful, constructs that allow the development of quite complex organisational structures. Thus, in Section 2, we introduce the concepts behind the new structures, in particular showing how they relate to common BDI language semantics. To clarify this further, in Section 3, we provide the core semantics of a subset of AGENTSPEAK incorporating the new concepts; we call this language AGENTSPEAK<sup>-</sup>. To show how these concepts can be used, in Section 4, we outline how a variety of organisational structures can be expressed using these simple constructs, and even provide implementations within AGENTSPEAK. Finally, in Section 5, we provide concluding remarks.

Thus, we begin by introducing the concepts; we do this by first considering the core operational aspects of BDI languages and then showing how our new concepts affect these.

## 2 Introducing the Concepts

Although all BDI languages have a family resemblance their syntax and semantics can vary immensely. We therefore use a loose unifying framework for our discussion into which we believe most BDI languages will fit<sup>1</sup>, although not always elegantly.

Our semantic framework assumes that a BDI language specifies the behaviour of an agent in terms of the agent's current state,  $S$ , which changes over time and a fixed specification,  $SP$ , which does not. Thus, an agent is viewed as a tuple  $\langle S, SP \rangle$ .  $S$  consists, amongst other things, of a set of beliefs,  $B$ . The BDI programming language then has a process for determining whether a given belief  $b$  follows from the current state which we will write as  $S \models b$ , since these are often logical mechanisms.

The BDI programming language has a specific operation, **select instruction**, which acts on the state according to the specification in order to determine the next instruction to be executed and another, **modify**, which modifies the state according to the specification

<sup>1</sup> Indeed, in [4] such a framework was used to provide a common semantic basis for 3APL, AGENTSPEAK, METATEM, etc.

and the selected instruction. The execution of an agent can therefore be viewed as repeated application of the transition rule

$$\langle \mathcal{S}, \mathcal{SP} \rangle \rightarrow \langle \text{modify}(\mathcal{SP}, \mathcal{S}, \text{select\_instruction}(\mathcal{SP}, \mathcal{S})), \mathcal{SP} \rangle . \quad (1)$$

We assume that both  $\mathcal{S}$  and  $\mathcal{SP}$  are made up of a number of sets or stacks (e.g., of beliefs) and use the notation  $\mathcal{S}[S_1/S_2]$  to indicate the state  $\mathcal{S}$  in which the set  $S_1$  has been replaced by  $S_2$ .

**Note.** This framework should not be interpreted as assuming that a given BDI language has explicit constructs for `select_instruction` and `modify`, but that most BDI languages can be expressed in terms of the operation of appropriate versions of these functions.

We also assume that a BDI language contains a set of plans (or rules),  $P$ , which are used by the `select_instruction` operation. These plans may either be a part of  $\mathcal{S}$  or  $\mathcal{SP}$ . We assume such plans are *triggered* in some fashion by  $\mathcal{S}$ . In some cases they are triggered by the composition of the Beliefs (e.g., METATEM [5]), in some by the Goals (e.g., 3APL [8, 3]) and in some by explicit trigger events (e.g., the *Jason* [2] interpreter for AGENTSPEAK [11]).

To simplify matters, we use an abstraction of a plan, describing it as:

$$t \leftarrow \{g\}b .$$

Thus, plans comprise; a trigger,  $t$ ; a guard (checked against an agent's beliefs),  $g$ ; and a Body,  $b$ , which specifies an instruction (or sequence of instructions) to be executed. In languages where only beliefs are used to trigger plans this can be written as

$$\top \leftarrow \{g\}b .$$

In order to trigger plans, the language requires some component of the current state  $\mathcal{S}$  which activates the trigger. We treat this as a set,  $T$ , and write the triggering process as  $T \models_t t$ .

Finally, we will use the notation  $Ag \models_a p$  to indicate that a plan,  $p$ , is applicable for an agent,  $Ag$ . The semantics of this for a basic BDI agent<sup>2</sup> is

$$\frac{\text{app\_cond}(t \leftarrow \{g\}b, Ag)}{Ag \models_a t \leftarrow \{g\}b} \quad (2)$$

where `app_cond` are the agent language's applicability conditions. In most languages

$$\text{app\_cond}(t \leftarrow \{g\}b, Ag) = T \models_t t \wedge \mathcal{S} \models g .$$

#### Notes.

- Again we do not necessarily expect these operations associated with plans to be explicit in the languages (e.g.,  $T$  may be a stack of goals and  $T \models_t g$  may be the process of matching the head (or prefix) of this stack).

<sup>2</sup> i.e., an agent whose semantics has not been modified with the constructs we describe later.

- There may be other applicability checking processes within the language (e.g., applicability of actions) — we represent all of these by  $Ag \models_a$ .
- Application of a plan results in an instruction to modify the state either directly (e.g.,  $+b$  appears in the body of the plan and is an instruction to add  $b$  to  $B$ ) or indirectly (e.g., the body of the plan is integrated into an *intention* or other part of the state which is then used for further planning or to govern subsequent actions and changes of belief).

Given the above basis, we now consider the two aspects we wish to introduce. The first, though related to the representation of groups of agents in METATEM [5], is independent of the underlying language for agents. The only restrictions we put on any underlying language is that, as in most BDI-based languages (and as described above), there are logical mechanisms for explicitly describing *beliefs* and *goals*, and possibly *plans* and *intentions*<sup>3</sup>.

## 2.1 Content and Context Sets

Assuming that the underlying language can describe the behaviour of an agent as above, we now extend the concept of agent with two sets, **Content** and **Context**. The agent's **Content** describes the set of agents it contains, while the agent's **Context** describes a set of agents it is contained within. The addition of **Content** and **Context** sets to each agent provides significant flexibility for agent organisation. Agent teams, groups or organisations, which might alternatively be seen as separate entities, are now just agents with non-empty **Content**. This allows these organisations to be hierarchical and dynamic, and so, as we will see later, provides possibilities for a multitude of other co-ordinated behaviours. Similarly, agents can have several agents within their **Context**. Not only does this allow agents to be part of several organisational structures simultaneously, but it allows the agent to benefit from **Context** representing diverse attributes/behaviours. So an agent might be in a context related to its physical locality (i.e. agents in that set are 'close' to each other), yet also might be in a context that provides certain roles or abilities. Intriguingly, agents can be within many, overlapping and diverse, contexts. This gives the ability to produce complex organisations, in a way similar to multiple inheritance in traditional object/concept systems. For some sample configurations, see Fig. 1.

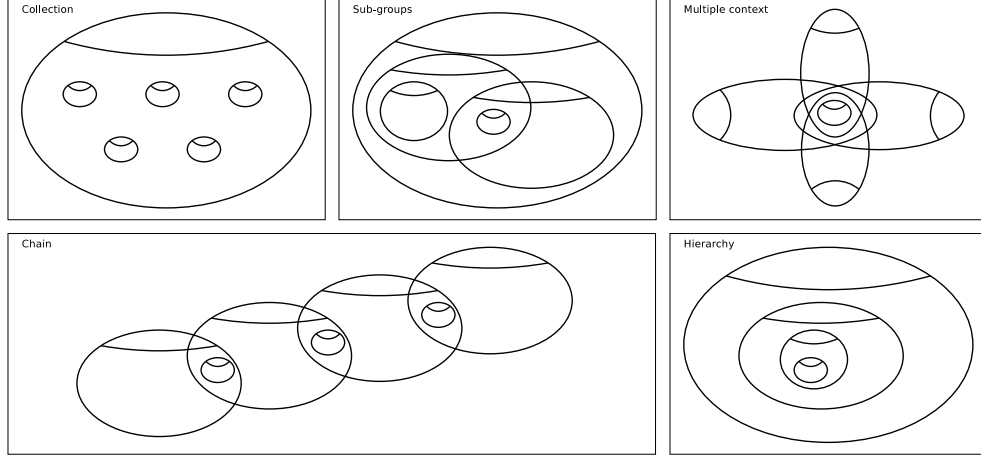
An important aspect is that this whole structure is very dynamic. Agents can move in and out of **Content** and **Context** sets, while new agents (and, hence, organisations) can be spawned easily and discarded. This allows for the possibility of a range of organisations, from the *transient* to the *permanent*. From the above it is clear that there is no enforced distinction between an agent and an agent organisation. All are agents, all may be treated similarly.

Finally, it is essential that the agent's internal behaviour, be it a program or a specification, have direct access to both the **Content** and **Context** sets. As we will see below, this allows each agent to become more than just a 'dumb' container. It can control access to, restructure, and share information and behaviours with, its **Content**.

## Semantics

The simplicity of the above approach allows us to provide a few general operational rules for managing the content and context.

<sup>3</sup> We also require that message-passing between agents is provided; this is standard in most languages.



**Fig. 1.** A selection of possible organisation structures.

We extend the agent's state,  $\mathcal{S}$ , with a content, ( $Cn$ ), and a context, ( $Cx$ ), and add four new instructions into the language  $+ag^{cn}$  (add  $ag$  to the content),  $-ag^{cn}$  (remove  $ag$  from the content) and  $+ag^{cx}$ ,  $-ag^{cx}$  for adding and removing agents from the context. We also add four new constructs into the trigger component,  $T$ :  $entered\_content(ag)$ ,  $entered\_context(ag)$ ,  $left\_content(ag)$  and  $left\_context(ag)$  and two new constructs into our language of guards:  $in\_content(ag)$  and  $in\_context(ag)$ .

We then extend the **modify** operation with the rules:

$$\mathbf{modify}(\mathcal{SP}, \mathcal{S}, +ag^{cn}) = \mathcal{S}[Cn/Cn \cup \{ag\}, T/T \cup entered\_content(ag)] \quad (3)$$

$$\mathbf{modify}(\mathcal{SP}, \mathcal{S}, -ag^{cn}) = \mathcal{S}[Cn/Cn/\{ag\}, T/T \cup left\_content(ag)] \quad (4)$$

and two analogous ones for context. These rules extend both the content/context and the trigger set,  $T$ . This allows plans to be triggered by changes in these sets. (e.g., plans of the form

$$entered\_content(Ag) \leftarrow \{in\_content(Ag)\}send(self, Ag, plan)$$

may be written which are triggered by the addition of a new agent  $Ag$  to the Content into sending that agent a plan).

We also extend the belief inference process to include checking membership of  $Cn$  and  $Cx$ :

$$\frac{ag \in Cn}{\mathcal{S} \models in\_content(ag)} \quad (5)$$

$$\frac{ag \in Cx}{\mathcal{S} \models in\_context(ag)} \quad (6)$$

It should be noted that in many languages it may be possible to streamline these extensions (e.g., by merging the triggering of plans and the update of content/context – see Section 3).

## 2.2 Constraints

The second basic component we suggest as being necessary for many meaningful multi-agent structures is that of *constraints*. A constraint consists of additional guards that may be appended to plans/rules and actions by an agent's context. This allows permissions to be modelled.

### Semantics

As with groups we extend the agent's state,  $\mathcal{S}$ , with a constraint set, ( $C$ ).  $C$  is treated as a set of pairs of a trigger and a guard, written  $[t \Rightarrow g]$ . Depending on the language, it may be desirable to add other pairs to this set, for instance if actions may have guards and there is an applicability process for actions then action/guard pairs may also be useful within constraints. Again, we add new instructions into the language  $+new\_constraint^c$  (add  $new\_constraint$  to  $C$ ) and  $-new\_constraint^c$  (remove  $new\_constraint$  from  $C$ ).

We then extend  $Ag \models_a$ :

$$\frac{\forall [t \Rightarrow g'] \in C. \mathcal{S} \models g' \quad \mathbf{app\_cond}(t \leftarrow \{g\}b, Ag)}{Ag \models_a t \leftarrow \{g\}b} \quad (7)$$

So, in many languages, this becomes

$$\frac{\forall [t \Rightarrow g'] \in C. \mathcal{S} \models g' \quad T \models_t t \quad \mathcal{S} \models g}{Ag \models_a t \leftarrow \{g\}b} \quad (8)$$

Similar modification can be made to the operational semantics of action applicability (internal or external) and any other relevant components of  $\mathcal{S}$  and  $\mathcal{SP}$ .

It should be noted that constraints make relatively little sense in a single agent environment (where guards on plans and actions are sufficient) it is only in a multi-agent environment where a context may wish to add additional guards to a pre-existing plan or action that such constraints become useful.

## 3 A Simple BDI Language

We will conclude our discussion of formal semantics with a simple example showing how our framework provides a practical methodology for extending existing BDI languages. Let us consider an extremely simple agent programming language based on AGENTSPEAK. We will call this language AGENTSPEAK<sup>-</sup>.

### 3.1 Syntax

Our language uses ground first-order formulae for beliefs, actions and goals. A plan is a triple of a goal, a guard and a stack of instructions (called here *deeds* following AIL [4]). An Agent is a triple of a set of beliefs, a stack of deeds and a set of plans. This is shown in Fig. 2.

$$\begin{aligned}
term &:= \text{Ground First Order Formula} \\
belief &:= term \\
action &:= term \\
goal &:= term \\
deed &:= action \mid +belief \mid -belief \mid +!goal \\
plan &:= goal : set(belief) \leftarrow stack(deed) \\
agent &:= \langle set(belief), stack(deed), set(plan) \rangle
\end{aligned}$$

**Fig. 2.** Syntax of AGENTSPEAK<sup>-</sup>

### 3.2 Operational Semantics

We provide a simple operational semantics for AGENTSPEAK<sup>-</sup> in the form of the four transition rules given in Fig. 3. In these semantics  $do(a)$  is an operation in an agent's interface that causes it to perform the action,  $a$ , and then returns a set of messages in the form of deeds,  $+!received(sender, \phi)$ , which instruct the agent to handle the message  $\phi$  from agent  $sender$ . In this language perception, therefore, has to be handled by an explicit *perception* action which then returns messages from the environment as if from another agent. Finally, ; represents the *cons* function on stacks, @ represents the join function, and *random* indicates random selection of an element from a set.

$$\frac{do(a) = msg}{\langle B, a; D, P \rangle \rightarrow \langle B, msg@D, P \rangle} \quad (9)$$

$$\frac{}{\langle B, +b; D, P \rangle \rightarrow \langle B \cup \{b\}, D, P \rangle} \quad (10)$$

$$\frac{}{\langle B, -b; D, P \rangle \rightarrow \langle B/\{b\}, D, P \rangle} \quad (11)$$

$$\frac{body = random(\{b \mid g : G \leftarrow b \in P \wedge G \subseteq B\})}{\langle B, +!g; D, P \rangle \rightarrow \langle B, body@D, P \rangle} \quad (12)$$

**Fig. 3.** Operational Semantics of AGENTSPEAK<sup>-</sup>

It should be noted that this is not intended as a practical example of a BDI language. For a start the language is entirely ground and makes no use of unification. Secondly the rather crude use of the deed stack to organise both planning and message handling/perception is likely to cause quite strange behaviour in any real agent setting.

### 3.3 Extension to the Simple BDI Language

Fig. 4 shows how this language fits into our earlier framework.

Framework	AGENTSPEAK <sup>-</sup>
$\mathcal{SP}$	$P$
$\mathcal{S}$	$B, D$
$T$	$D$
$\mathcal{S} \models b$	$b \subseteq B$
$T \models_t t$	$t = hd(D)$
<b>app_cond</b> ( $gl : g \leftarrow b$ )	$g \subseteq B$
<b>modify</b> (( $B, D$ ), $P$ , $+b$ )	$(B \cup \{b\}, D)$
<b>modify</b> (( $B, D$ ), $P$ , $-b$ )	$(B/\{b\}, D)$
<b>modify</b> (( $B, D$ ), $P$ , $ds$ )	$(B, ds@D)$
<b>select_instruction</b> (( $B, a; D$ ), $P$ )	$do(a)$
<b>select_instruction</b> (( $B, +b; D$ ), $P$ )	$+b$
<b>select_instruction</b> (( $B, -b; D$ ), $P$ )	$-b$
<b>select_instruction</b> (( $B, +!g; D$ ), $P$ )	$random(\{b \mid p \in P \wedge Ag \models_a p\})$

**Fig. 4.** Mapping our Framework to AGENTSPEAK<sup>-</sup>

Modifying these semantics according to our content/context/constraints framework now gives us the language semantics shown in Fig. 5

In fact this extension can be improved upon based on the details of our languages. For instance we can omit the *entered\_content()* and *left\_content()* and use  $+ag^{cn}$  and  $-ag^{cn}$  as plan triggers if we like changing (19) to

$$\frac{body = random(\{b \mid +ag^{cn} : G \leftarrow b \in P \wedge G \subseteq B \wedge \forall [+ag^{cn} \Rightarrow G'] \in C. G' \subseteq B\}}{\langle B, +ag^{cn}; D, Cn, Cx, C, P \rangle \rightarrow \langle B, body@D, Cn \cup \{ag\}, Cx, C, P \rangle} \quad (23)$$

## 4 Using the Concepts

We will briefly discuss some illustrative examples of the use of constraints and content/context sets (sometimes termed *groups*) in organisational settings.

### 4.1 Shared beliefs

Being a member of all but the least cohesive groups/organisations requires that some shared beliefs exist between its members. Making the contentious assumption that all agents are honest and that joining a group is both individual rational and group rational, let agent  $i$  hold a belief set  $BS_i$  and assume the programming language contains the instruction *addBelief*(*Beliefs*) with the semantics

$$\mathbf{modify}(\mathcal{SP}, \mathcal{S}, \mathbf{addBelief}(Bs)) = \mathcal{S}[B/B \cup Bs]$$

Suppose a (group) agent  $i$  has the plan:

$$\mathit{entered\_content}(Ag) \leftarrow \{\mathit{in\_content}(Ag)\}\mathit{send}(i, Ag, \mathit{inform}(BS_i))$$

$$\frac{do(a) = msg \quad \forall [a \Rightarrow G] \in C. G \subseteq B}{\langle B, a; D, Cn, Cx, C, P \rangle \rightarrow \langle B, msg@D, Cn, Cx, C, P \rangle} \quad (13)$$

$$\frac{}{\langle B, +b; D, Cn, Cx, C, P \rangle \rightarrow \langle B \cup \{b\}, D, Cn, Cx, C, P \rangle} \quad (14)$$

$$\frac{}{\langle B, -b; D, Cn, Cx, C, P \rangle \rightarrow \langle B/\{b\}, D, Cn, Cx, C, P \rangle} \quad (15)$$

$$\frac{}{\langle B, +c^c; D, Cn, Cx, C, P \rangle \rightarrow \langle B, D, Cn, Cx, C \cup \{c\}, P \rangle} \quad (16)$$

$$\frac{}{\langle B, -c^c; D, Cn, Cx, C, P \rangle \rightarrow \langle B, D, Cn, Cx, C/\{c\}, P \rangle} \quad (17)$$

$$\frac{body = random(\{b \mid g : G \leftarrow b \in P \wedge G \subseteq B \wedge \forall CgG' \in C. G' \subseteq B\})}{\langle B, +!g; D, Cn, Cx, C, P \rangle \rightarrow \langle B, body@D, Cn, Cx, C, P \rangle} \quad (18)$$

$$\frac{}{\langle B, +ag^{cn}; D, Cn, Cx, C, P \rangle \rightarrow \langle B, +!entered\_content(ag); D, Cn \cup \{ag\}, Cx, C, P \rangle} \quad (19)$$

$$\frac{}{\langle B, -ag^{cn}; D, Cn, Cx, C, P \rangle \rightarrow \langle B, +!left\_content(ag); D, Cn/\{ag\}, Cx, C, P \rangle} \quad (20)$$

$$\frac{}{\langle B, +ag^{cx}; D, Cn, Cx, C, P \rangle \rightarrow \langle B, +!entered\_context(ag); D, Cn, Cx \cup \{ag\}, C, P \rangle} \quad (21)$$

$$\frac{}{\langle B, -ag^{cx}; D, Cn, Cx, C, P \rangle \rightarrow \langle B, +!left\_context(ag); D, Cn, Cx/\{ag\}, C, P \rangle} \quad (22)$$

**Fig. 5.** AGENTSPEAK<sup>-</sup> extended to multi-agents

and agent  $j$  has the plan:

$$received(Ag, j, inform(BS_i)) \leftarrow \{in\_context(Ag)\}addBelief(BS_i)$$

taken together these plans mean that if  $j$  joins the Content of  $i$  it gets sent the beliefs  $BS_i$  which it adds to its own belief base. This allows shared beliefs to be established.

The agent in receipt of the new beliefs may or may not disseminate them to the agents in its Content, depending on the nature and purpose of the group. Once held, beliefs are retained until contradicted or revised (for example, on leaving the group).

## 4.2 Permissions and Obligations

A number of multi-agent formalisms include concepts of permissions and obligations. An agent within a group setting may or may not have the permission to perform a particular action or communicate in a particular fashion. This can be easily represented using constraints: for instance if agents in group,  $G$ , may not perform action  $a$  then the constraint  $[a \Rightarrow \perp]$  can be communicated to them when they join  $G$ 's Content.

It should be noted that in order for such a message to be converted into an actual constraint, the receiving agent would also need the plan:

$$received(Ag, i, constrain([a \Rightarrow g])) \leftarrow \{in\_context(Ag)\} + [a \Rightarrow g]^c$$

This design deliberately allows varying degrees of autonomy among agents to be handled by the programmer.

Obligations are where a group member is obliged to behave in a particular fashion. This can be modelled if plans are treated as modifiable by the underlying agent language. Obligations can then be communicated as new plans.

```

/*----- initial beliefs ----- */
cooperative.

/*----- rules ----- */
check_constraint(Plan, Arg)
:- not constraint_fails(Plan,Arg).

/* ----- basic plans ----- */

/* How an agent responds to a group membership invitations */
+!join(Group)[source(Group)] : cooperative
<- .my_name(Me);
+context(Group);
.println("I believe I have the context of ", Group);
.send(Group, achieve, accept(Me, Group)).

```

**Fig. 6.** A simple cooperative agent defined in AgentSpeak

### 4.3 Pizzas

Our last example is a simple case study which we have implemented in AgentSpeak/Jason. It concerns a simple *cook* agent who is provided with a number of plans by a chef agent, each for cooking a different meal. The cook's choice of plan is constrained by the Context in which it cooks.

**Scenario.** The chef of a restaurant hires a cook and provides a list of dishes from which the cook is free to choose when asked to prepare a meal. As diners arrive, their preferences are noted and the cook endeavours to choose a meal that satisfies all of the diners.

Our cook was implemented as a simple, cooperative agent with the ability to enter the Context of other agents but without any domain abilities — it can't cook — see Fig. 6.

When hired, the cook agent receives plans for making risotto, steak and pizza. AgentSpeak code defining this behaviour is shown in Fig. 7. When asked to prepare a meal without

```
+content(Agent)[source(self)]
  <- .print("Sending ", Agent, " plans...");
    .send(Agent, tellHow, "+!cook(risotto) : check_constraint(cook,risotto)
                                <- make(risotto).");
    .send(Agent, tellHow, "+!cook(steak) : check_constraint(cook,steak)
                                <- make(steak).");
    .send(Agent, tellHow, "+!cook(pizza) : check_constraint(cook,pizza)
                                <- make(pizza).").
```

Fig. 7. When an agent is 'hired' by a Chef it divulges plans

the constraints of any diners it prepares risotto; see Fig. 9(b). A meat eating diner then imposes their dislike for risotto by providing the cook with the constraint

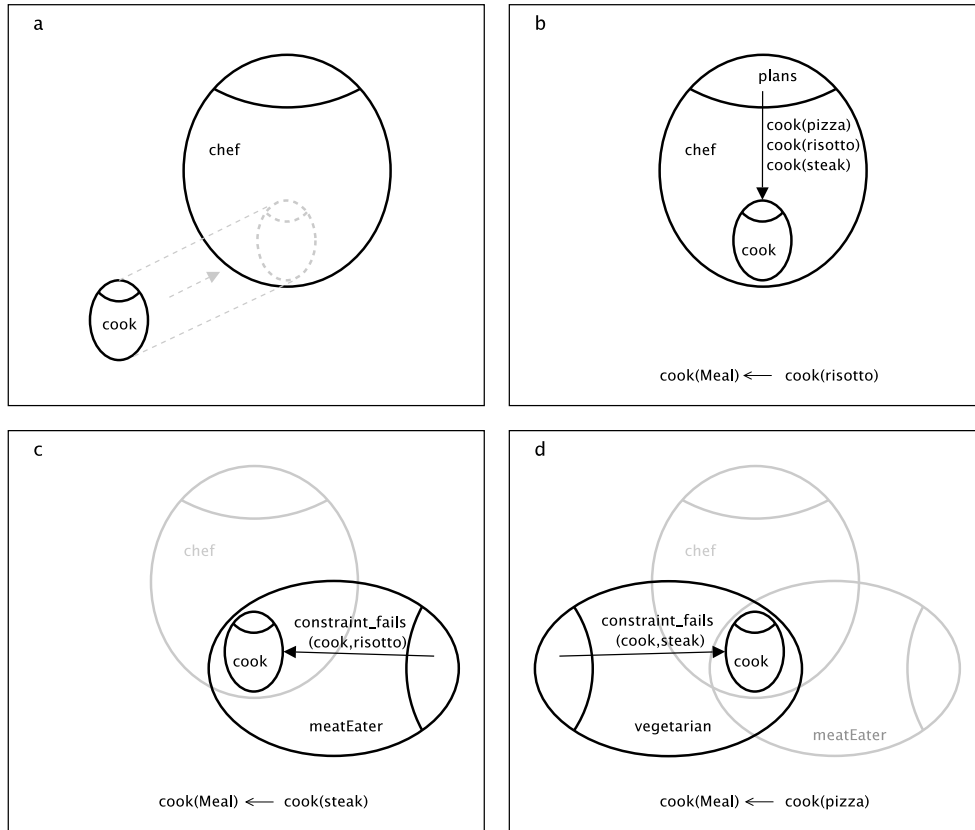
```
constraint_fails(cook,risotto).
```

Now, acting in the context of this meat eater, the chef prepares steak as opposed to risotto; see Fig. 9(c). Finally, a vegetarian diner also invites the chef agent to join its Content and in so doing imposes the constraint `constraint_fails(cook,steak)`, see Fig. 8. The agent, now a member of three contexts must decide an appropriate course of action within the supplied constraints - it must not commit to cooking risotto or steak! Thus it is constrained to choose to prepare pizza; see Fig. 9(d).

```
+content(Agent)[source(self)]
  <- .print("Sending ", Agent, " my constraints");
    .send(Agent, tell, constraint_fails(cook,steak)).
```

Fig. 8. Vegetarians don't eat steak.

Full execution output for this example is given in Fig. 10.



**Fig. 9.** A cook with multiple constraints.

## 5 Concluding Remarks

In this paper we have proposed a simple extension to BDI languages that permits the development of sophisticated multi-agent organisations. We have shown how the addition of both content/context sets and constraints is both semantically simple and appealing. Although we provided a semantic definition for a simple BDI language, we gave this only for illustrative purposes. We expect that developers' favourite logical agent language could be extended in this way.

Finally, we provided some simple examples to justify our statement that many agent organisational aspects can be modelling using our two simple concepts. While we were not able to give more comprehensive justification, we note that, in a companion paper [7], we show how leading work on organisations, roles and teamwork can all fit within our framework.

```

[chef] saying: inviting cook to join my content
[cook] saying: I believe I have the context of chef
[chef] saying: Sending cook plans...
[chef] saying: I consider cook to be in my content
[cook] doing: make(risotto)
***cook is making risotto***
[meatEater] saying: inviting cook to join my content
[cook] saying: I believe I have the context of meatEater
[meatEater] saying: Sending cook my constraints
[meatEater] saying: I consider cook to be a member of my content
[cook] doing: make(steak)
***cook is making steak***
[vegetarian] saying: inviting cook to join my group
[cook] saying: I believe I have the context of vegetarian
[vegetarian] saying: Sending cook my constraints
[vegetarian] saying: I consider cook to be in my content
[cook] doing: make(pizza)
***cook is making pizza***

```

**Fig. 10.** Execution output

## References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.
2. R. H. Bordini, J. F. Hübner, and R. Vieira. *Jason* and the golden fleece of agent-oriented programming. In Bordini et al. [1], chapter 1, pages 3–37.
3. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming Multi-Agent Systems in 3APL. In Bordini et al. [1].
4. L. A. Dennis, B. Farwer, R. H. Bordini, M. Fisher, and M. Wooldridge. A Common Semantic Basis for BDI Languages. In *Proc. Seventh International Workshop on Programming Multiagent Systems (ProMAS)*, Lecture Notes in Artificial Intelligence. Springer Verlag, 2007 (to appear).
5. M. Fisher. METATEM: The Story so Far. In *Proc. Third International Workshop on Programming Multiagent Systems (ProMAS)*, volume 3862 of *Lecture Notes in Artificial Intelligence*, pages 3–22. Springer Verlag, 2006.
6. M. Fisher, E. Pearce, M. Wooldridge, M. Sierhuis, W. Visser, and R. Bordini. Towards the Verification of Human-Robot Teams. In *IEEE Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA)*, Washington D.C., USA., 2005.
7. A. J. Hepple, L. A. Dennis, and M. Fisher. Building Agent Organisations in BDI Languages the Simple Way. Submitted., 2007.
8. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Agent Programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
9. N. R. Jennings and M. Wooldridge. Applications of agent technology. In *Agent Technology: Foundations, Applications, and Markets*. Springer-Verlag, Heidelberg, 1998.
10. N. Muscettola, P. P. Nayak, B. Pell, and B. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.

11. A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In *Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer, 1996.
12. A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.
13. M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See <http://ic.arc.nasa.gov/ic/publications>), 2006.
14. M. Sierhuis, J. M. Bradshaw, A. Acquisti, R. V. Hoof, R. Jeffers, and A. Uszok. Human-Agent Teamwork and Adjustable Autonomy in Practice. In *Proceedings of the 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS)*, Nara, Japan, 2003.
15. M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.