In Multi-Agent Systems Methodologies and Applications: Proceedings of the Second Australian Workshop on Distributed Artificial Intelligence, C. Zhang and D. Lukose, (eds.), Lecture Notes in Artificial Intelligence, 1286, 48-62, Springer-Verlag, 1997

Making and Breaking Engagements: An Operational Analysis of Agent Relationships

Mark d'Inverno[1] and Michael Luck[2]

[1] School of Computer Science, University of Westminster, London, W1M 8JS, UK.
Email: dinverm@westminster.ac.uk
[2] Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK.
Email: mikeluck@dcs.warwick.ac.uk

Abstract. Fundamental to the operation of multi-agent systems is the concept of cooperation between individual agents by which the overall system exhibits significantly greater functionality than the individual components. Not only is it important to understand the structure of such relationships in multi-agent systems, it is also important to understand the ways in which cooperative relationships come about. This is particularly true if such an analysis is to be relevant to, and useful for, the construction of real systems for which the invocation and destruction of such relationships is critical. This paper extends previous work concerned with formalising the structure of inter-agent relationships to provide an operational analysis of the invocation and destruction of engagement and cooperation within our formal agent framework.

1 Introduction

Fundamental to the operation of multi-agent systems is the concept of cooperation between individual agents. If single agent systems can cooperate, they can exploit the capabilities and functionality of others to achieve their own individual goals. This moves beyond the advantages of robustness in traditional distributed systems in the face of individual component failure since components can be replaced and cooperation configurations realigned. It allows the specific expertise and competence of different agents to complement each other so that in addition to general resilience, the overall system exhibits significantly greater functionality than individual components.

Durfee, for example, defines a multi-agent system as a collection of problem solvers that "work together" to achieve goals that are beyond the scope of their individual abilities [2]. This notion of agents helping each other, working together or cooperating in some way is common. Huberman and Clearwater similarly characterise multi-agent systems by the interaction of many agents trying to solve problems in a cooperative fashion [3], and Lesser describes a multi-agent system as a computational system in which several semi-autonomous agents interact or work together to perform some tasks or satisfy some goals [4].

Since cooperation underlies the structure of multi-agent systems, it is important to be able to model and anlayse it in detail in the context of a well-founded framework. Moreover, the ways in which cooperative relationships come about

are also important for a complete understanding of the nature of cooperation. This is especially true if such an analysis is to be relevant to real systems for which the invocation and destruction of such relationships is critical.

Previous work has focussed on taxonomising the types of interactions that occur between agents in a multi-agent system, distinguishing in particular between *engagements* of non-autonomous agents and *cooperation* between autonomous agents [7]. This work was based on our agent hierarchy which defined *agency* and *autonomy*, and specified how autonomy is distinct but is achieved by *motivating* agency and generating goals [5]. In this paper we aim to extend this work on cooperation and engagement by describing the operations necessary for their invocation and destruction, and providing a formal specification in the context of the existing framework. First, we briefly review the agent framework and the nature of engagement and cooperation within it. We then address the invocation and destruction of such structures, enumerating the variety of situations in which they arise. Finally, we discuss what benefits such an analysis provides.

2 Background

To present a mathematical description of our definitions, we use the specification language, Z [8]. Space constraints prohibit a detailed explanation of the notation here, but our use should be clear from the combination of text and formalism. (A brief introduction to Z can be found in the appendix of [1].) In this section, we sketch previous work on the agent framework. Details may be found in [5, 7].

An *object* comprises a set of *motivations*, a set of *goals*, a set of *actions*, and a set of *attributes* such that the attributes and actions are non-empty.

$$
\begin{array}{|l}
\textit{Object} \\
\hline
attributes : \mathbb{P}\ Attribute \\
capableof : \mathbb{P}\ Action \\
goals : \mathbb{P}\ Goal \\
motivations : \mathbb{P}\ Motivation \\
\hline
attributes \neq \{\} \land capableof \neq \{\} \\
\hline
\end{array}
$$

An object, described by its features and capabilities, is just an automaton and cannot engage other entities to perform tasks, nor is it itself used in this way. However, it serves as a useful abstraction mechanism by which it is regarded as distinct from the remainder of the environment, and can subsequently be transformed into an agent, an augmented object, engaged to perform some task or satisfy some goal. Viewing something as an agent means that we regard it as satisfying or pursuing some goal. Furthermore, it means that it must be doing so either for another agent, or for itself, in which case it is autonomous. If it is for itself, it must have generated the goal through its motivations. Autonomous agents are thus just agents that can generate their own goals from motivations.

Agent

$Object$

$goals \neq \{\}$

AutonomousAgent

$Agent$

$motivations \neq \{\}$

For ease of exposition, we further refine the agent hierarchy with two new definitions. A *neutral-object* is an object that is *not* an agent, and a *server-agent* is an agent that is *not* an autonomous agent.

NeutralObject

$Object$

$goals = \{\} \land motivations = \{\}$

ServerAgent

$Agent$

$motivations = \{\}$

Now, we consider a multi-agent system as a collection of objects, agents and autonomous as defined below. In this view, a multi-agent system contains autonomous agents which are all agents, and in turn, all agents are objects. An agent is either a server-agent or an autonomous agent, and an object is either a neutral-object or an agent.

MultiAgentSystemComponents

$objects : \mathbb{P}\ Object$
$agents : \mathbb{P}\ Agent$
$autonomousagents : \mathbb{P}\ AutonomousAgent$
$neutralobjects : \mathbb{P}\ NeutralObject$
$serveragents : \mathbb{P}\ ServerAgent$

$autonomousagents \subseteq agents \subseteq objects$
$agents = autonomousagents \cup serveragents$
$objects = neutralobjects \cup agents$

3 Engagement and Cooperation

In this section, we briefly review previous work that categorised inter-agent relationships into two distinct classes, engagements and cooperations. We provide formal descriptions and brief explanations. For a full exposition on the nature of these relationships, see [7]. Note, however, that there have been several minor modifications to the earlier formal specification.

3.1 Engagement

A direct engagement occurs when a neutral-object or a server-agent adopts some goals. In a direct engagement, a *client*-agent with some goals uses another *server*-agent to assist them in the achievement of those goals. A server-agent either exists already as a result of another engagement, or is instantiated from a neutral-object for the current engagement. No restriction is placed on a client-agent.

We define a *direct engagement* below to consist of a client agent, *client*, a server agent, *server*, and the goal that *server* is satisfying for *client*. An agent cannot engage itself, and both agents must have the goal of the engagement.

DirectEngagement

$$client : Agent$$
$$server : ServerAgent$$
$$goal : Goal$$

$$client \neq server$$
$$goal \in (client.goals \cap server.goals)$$

The set of all *direct* engagements in a system is given by *direngagements* in the following schema. For any direct engagement in *direngagements*, there can be no intermediate *direct* engagements of the goal, so there is no other agent, y, where *client* engages y for *goal*, and y engages *server* for *goal*.

SystemEngagements

$$MultiAgentSystemComponents$$
$$direngagements : \mathbb{P}\ DirectEngagement$$

$$\forall eng : direngagements \bullet \neg (\exists y : Agent; e_1, e_2 : direngagements \mid e_1.goal = e_2.goal = eng.goal \bullet e_1.server = eng.server \wedge e_2.client = eng.client \wedge e_1.client = e_2.server = y)$$

An *engagement chain* represents a sequence of *direct engagements*. Specifically, an *engagement chain* comprises a goal, *goal*, the autonomous client that generated the goal, *autoagent*, and a sequence of server-agents, *agentchain*, where each agent in the sequence directly engages the next. For any engagement chain, there must be at least one server-agent, all agents must share *goal*, and each agent can only be involved once. seq_1 represents a non-empty sequence.

EngagementChain

$$goal : Goal$$
$$autoagent : AutonomousAgent$$
$$agentchain : \mathsf{seq}_1\ Agent$$

$$goal \in autoagent.goals$$
$$goal \in \bigcup \{s : Agent \mid s \in \mathsf{ran}\ agentchain \bullet s.goals\}$$
$$\#(\mathsf{ran}\ agentchain) = \#agentchain$$

The set of all engagement chains in a system is given in the schema below by *engchains*. For every engagement chain, *ec*, there must be a direct engagement between the autonomous agent, $ec.autoagent$, and the first client of *ec*, $head\ ec.agentchain$, with respect to the goal of *ec*, $ec.goal$. There must also be a direct engagement between any two agents which follow each other in $ec.agentchain$ with respect to $ec.goal$. In general, an agent *engages* another agent if there is some engagement chain in which it precedes the server agent.

SystemEngagementChains

$$
\begin{array}{l}
SystemEngagements \\
engchains : \mathbb{P}\ EngagementChain \\
\hline
\forall\ ec : engchains;\ s_1, s_2 : Agent \bullet \\
\quad (\exists\ d : direngagements \bullet d.goal = ec.goal \land d.client = ec.autoagent \\
\qquad\qquad \land\ d.server = head\ ec.agentchain) \land \\
\quad \langle s_1, s_2 \rangle \ \mathsf{in}\ ec.agentchain \Rightarrow (\exists\ d : direngagements \bullet \\
\qquad\qquad d.client = s_1 \land d.server = s_2 \land d.goal = ec.goal)
\end{array}
$$

3.2 Cooperation

Two autonomous agents are said to be *cooperating* with respect to some goal if one of the agents has adopted goals of the other. This notion of autonomous goal acquisition applies both to the *origination* of goals by an autonomous agent for its own purposes, and the *adoption* of goals from others, since in each case the goal must have a positive motivational effect [6]. For autonomous agents, the goal of another can only be adopted if it has such an effect, and this is also exactly why and how goals are originated. Thus goal adoption and origination are related forms of goal generation. The term *cooperation* can be used only when those involved are autonomous and, potentially, capable of resisting. If they are not autonomous, nor capable of resisting, then one simply *engages* the other.

A *cooperation* describes a goal, the autonomous agent that generated the goal, and those autonomous agents who have adopted that goal from the generating agent. In addition, all the agents involved have the goal of the cooperation, an agent cannot cooperate with itself, and the set of cooperating agents must be non-empty. Cooperation cannot, therefore, occur unwittingly between agents, but must arise as a result of the motivations of an agent and that agent recognising the goal in another.

Cooperation

$$
\begin{array}{l}
goal : Goal \\
generatingagent : AutonomousAgent \\
cooperatingagents : \mathbb{P}\ AutonomousAgent \\
\hline
goal \in generatingagent.goals \\
\forall\ aa : cooperatingagents \bullet goal \in aa.goals \\
generatingagent \notin cooperatingagents \\
cooperatingagents \neq \{\ \}
\end{array}
$$

The set of cooperations in a multi-agent system is given by the *cooperations* variable in the schema, *SystemCooperations*. The predicate part of the schema states that for any cooperation, the union of the cooperating agents and the generating agent is a *subset* of the set of all autonomous agents which have that goal. As a consequence, two agents sharing a goal are not necessarily cooperating. In addition, the set of all cooperating agents is a subset of all autonomous agents of the system since not all are necessarily participating in cooperations.

SystemCooperations

$$
\begin{array}{l}
MultiAgentSystemComponents \\
cooperations : \mathbb{P}\ Cooperation \\
\hline
\forall\, c : cooperations \bullet (c.cooperatingagents \cup \{c.generatingagent\}) \subseteq \\
\qquad\qquad \{a : autonomousagents \mid c.goal \in a.goals \bullet a\} \\
\bigcup \{c : cooperations \bullet c.cooperatingagents\} \subseteq autonomousagents
\end{array}
$$

4 Operational Aspects of Engagement and Cooperation

So far we have provided an analysis of the *structure* of multi-agent systems based on inter-agent relationships. In this section, we provide an operational analysis of how these cooperations and direct engagements are created and destroyed, and how this affects the configuration of inter-agent relationships.

There are four principal operations to be considered, as follows:

- a server-agent adopting the goals of an agent giving rise to a new engagement;
- a server-agent being released from some or all of its agency obligations by an engaging agent, thus destroying a direct engagement;
- an autonomous agent adopting the goal of another, so that either a new cooperation is formed, or the agent joins an existing cooperation;
- an autonomous agent destroying the goals of an existing cooperation, resulting in either the destruction of the cooperation, or the removal of the autonomous agent from the cooperation.

To provide an operational account of these relationships we must specify how they are affected when new cooperations and engagements are invoked and destroyed. Before considering the operations in detail we first specify some general functions to create relationships from individual components. Thus, *MakeEng*, *MakeEngChain* and *MakeCoop* below simply construct the schema types, *Engagment*, *EngagementChain* and *Cooperation* respectively. The functions, *MakeEng* and *MakeCoop*, are partial; *MakeEng* is not defined if the two agents involved in the cooperation are the same, and *MakeCoop* is not defined if the single autonomous agent initiating the cooperation is in the set of other autonomous agents. Note that $\mathbb{P}_1\ AutonomousAgent$, represents non-empty sets

of elements of type *AutonomousAgents*. The schema also makes use of the *mu-expression*. In general, the value of a in the mu-expression, $\mu\, a : A \mid p$, is the unique value of type A which satisfies predicate p. [3]

$$
\begin{array}{|l}
MakeEng : (Goal \times Agent \times ServerAgent) \pfun DirectEngagement \\
MakeEngChain : (Goal \times AutonomousAgent \times \mathsf{seq}_1\, ServerAgent) \\
\qquad\qquad \pfun EngagementChain \\
MakeCoop : (Goal \times AutonomousAgent \times \mathbb{P}_1\, AutonomousAgent) \\
\qquad\qquad \fun Cooperation \\
\hline
\forall\, g : Goal; a : Agent; aa : AutonomousAgent; s : ServerAgent; \\
\qquad ch : \mathsf{seq}_1\, ServerAgent; aas : \mathbb{P}_1\, AutonomousAgent \bullet \\
\qquad MakeEng(g, a, s) = (\mu\, d : DirectEngagement \mid \\
\qquad\quad d.goal = g \land d.client = a \land d.server = s) \land \\
\qquad MakeEngChain(g, a, ch) = (\mu\, ec : EngagementChain \mid \\
\qquad\quad ec.goal = g \land ec.autoagent = a \land ec.agentchain = ch) \land \\
\qquad MakeCoop(g, aa, aas) = (\mu\, c : Cooperation \mid \\
\qquad\quad c.goal = g \land c.generatingagent = aa \land c.cooperatingagents = aas)
\end{array}
$$

Next, we define the generic function, *cut*, which takes an injective sequence (where no element appears more than once) and an element, and removes all the elements of the sequence which appear after this element. If the element does not exist in the sequence, the sequence is left unaltered.

$$
\begin{array}{|l}
[X] \\
\hline
cut : (\mathsf{iseq}\, X \times X) \pfun \mathsf{iseq}\, X \\
\hline
\forall\, x : X; s, t : \mathsf{seq}\, X \bullet cut(s, x) = t \Leftrightarrow \\
\qquad last\, t = x \land (\exists\, u : \mathsf{seq}\, X \bullet s = t \frown u) \\
\hline
\end{array}
$$

We also define a total function that creates a new object by ascribing a set of goals to an existing object. It is valid for any object and any set of goals. This is detailed elsewhere with a complete formalisation of goal generation [6].

$$
\begin{array}{|l}
ObjectAdoptGoals : (Object \times \mathbb{P}\, Goal) \fun Object \\
\hline
\forall\, gs : \mathbb{P}\, Goal; old, new : Object \bullet \\
\qquad ObjectAdoptGoals(old, gs) = new \Leftrightarrow new.goals = old.goals \cup gs \\
\qquad\quad \land\, new.capableof = old.capableof \land new.attributes = old.attributes
\end{array}
$$

This allows us to define two further functions, *ExtendChain* and *CutChain*. The first takes an engagement chain and an object, and extends the engagement chain to include the object. The second function cuts an engagement chain after the occurrence of an object.

[3] For example, the expression $\mu\, n : \mathbb{N} \mid (n * n) \bmod 2 = 0 \land 0 < (n * n) \leq 10l$ binds the variable n to the value 10.

$$
\begin{array}{l}
ExtendChain : EngagementChain \times Object \longrightarrow EngagementChain \\
CutChain : EngagementChain \times Agent \nrightarrow EngagementChain \\
\hline
\forall\, c : EngagementChain; e : Object \bullet \\
\quad ExtendChain(c, e) = (\mu\, new : EngagementChain \mid \\
\qquad new.goal = c.goal \wedge new.autoagent = c.autoagent \\
\qquad \wedge\, new.agentchain = c.agentchain^{\frown} \\
\qquad\qquad \langle ObjectAdoptGoals(e, \{c.goal\})\rangle \wedge \\
\quad CutChain(c, e) = (\mu\, new : EngagementChain \mid \\
\qquad new.goal = c.goal \wedge new.autoagent = c.autoagent \\
\qquad \wedge\, new.agentchain = cut(c.agentchain, e)))
\end{array}
$$

Formally, the structure of multi-agent systems is defined below. It states that a multi-agent system must contain at least one relationship between two agents.

$$
\begin{array}{l}
\underline{\quad MultiAgentSystemStructure \quad} \\
MultiAgentSystemComponents \\
SystemEngagements \\
SystemEngagementChains \\
SystemCooperations \\
\hline
\#cooperations + \#direngagements \geq 1 \\
\hline
\end{array}
$$

4.1 Making Engagements

When a new direct engagement is formed between an agent and a server agent, the associated engagement chain may be altered in several ways. The different possibilities depend on whether the engaging agent is at the tail, the head, or in the middle of the chain. Consider an engagement chain where A is an autonomous agent at the head of the chain directly engaging the server-agent, $S1$, which is directly engaging server-agent, *S2*, in turn directly engaging *S3*, as in Figure 1(a).

- If the last agent in the chain, $S3$, engages a neutral-object, O, the chain is extended to include the engagement between $S3$ and O, as in Figure 1(b).
- If the autonomous agent, A, directly engages O, a new engagement chain is created solely comprising A and O, as in Figure 1(c).
- If any *server*-agent, other than that at the tail of the engagement chain, engages O, then a new engagement chain is formed between them. Thus if $S1$ engages O, the existing chain is unchanged, but a new chain is formed from the engagements up to and including $S1$ in the original chain, with the addition of the new engagment of O by $S1$, as in Figure 1(d).

The aspects of forming a new engagement common to all three scenarios are described in the next schema. Here, the engaging agent, *agent*?, the engaged object, *e*?, the goal of the engagement, *goal*?, and an optional engagement chain, *chain*?, are given as input to the operation, and the structure of the multi-agent system changes. The predicate part states that the object is a system object, and the agent is a known agent with the goal, *goal*?. If no engagement

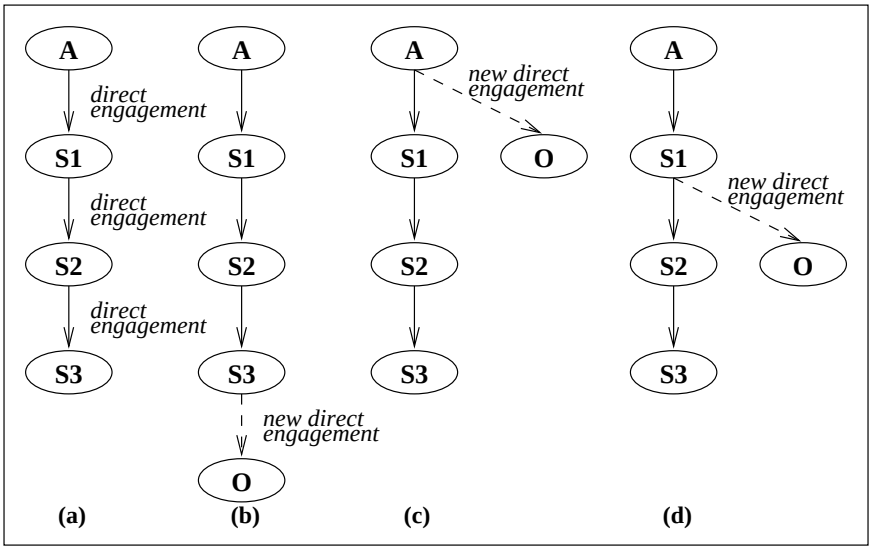

Fig. 1. Alternative ways to make engagements

chain already exists, so that *chain*? is not defined, then *agent*? must be autonomous. Conversely, if *chain*? is defined, *agent*? must be a server-agent, the goal of *chain*? must be *goal*?, and *agent*? must be part of *chain*?. (Formal definitions for optional, defined, undefined and the are given in the appendix.) There is no change to the set of cooperations, but the set of direct engagements is updated to include the new engagement between the agent and the object.

GeneralEngage

$agent? : Agent$
$e? : Object$
$goal? : Goal$
$chain? : \mathsf{optional}\ [EngagementChain]$
$\Delta MultiAgentSystemStructure$

$e? \in objects$
$agent? \in agents$
$goal? \in agent?.goals$
$\mathsf{undefined}\ chain? \Leftrightarrow agent? \in autonomousagents$
$\mathsf{defined}\ chain? \Rightarrow$
$\quad (agent? \in serveragents \land$
$\quad (\mathsf{the}\ chain?).goal = goal? \land$
$\quad agent? \in \mathsf{ran}\ (\mathsf{the}\ chain?).agentchain)$
$cooperations' = cooperations$
$direngagements' = direngagements \cup$
$\quad \{MakeEng(goal?, agent?, (ObjectAdoptGoals\,(e?, \{goal?\})))\}$

The distinct aspects of the ways in which the set of engagement chains are affected in each scenario are detailed below. First, the engaging agent is a server-agent at the end of the chain so that the chain is extended to include this new direct engagement.

EngageExtendChain

GeneralEngage

$$\mathsf{defined}\ chain? \land e? = last(\mathsf{the}\ chain?).agentchain \Rightarrow$$
$$engchains' = engchains \setminus chain? \cup$$
$$\{ExtendChain((\mathsf{the}\ chain?), e?)\}$$

Second, the engaging agent is autonomous, and a new engagement chain is formed from *goal*?, *agent*?, and the sequence consisting of *agent*? and the newly instantiated agent.

StartNewChain

GeneralEngage

$$agent? \in autonomousagents \Rightarrow$$
$$engchains' = engchains \cup$$
$$\{MakeEngChain(goal?, agent?, \langle agent?, ObjectAdoptGoals(e?, \{goal?\})\rangle)\}$$

Third, if *agent*? is not at head or tail of the chain, then the original chain is unchanged, and a new chain is formed from the direct engagements in the original chain up to the agent, plus the new direct engagement between *agent*? and the newly instantiated object.

CutandAddChain

GeneralEngage

$$(\mathsf{defined}\ chain?) \land (e? \neq last(\mathsf{the}\ chain?).agentchain)$$
$$\land\ e? \neq (head(\mathsf{the}\ chain?).agentchain) \Rightarrow$$
$$engchains' = engchains \cup$$
$$\{ExtendChain((CutChain((\mathsf{the}\ chain?), agent?)), e?)\}$$

The operation of making an engagement can then be defined using schema disjunction. Thus, the *Engage* operation is applied when any of the following three occur: *CutandAddChain*, *EngageExtendChain* or *StartNewChain*.

$$Engage \mathrel{\widehat{=}} EngageExtendChain \lor StartNewChain \lor CutandAddChain$$

4.2 Breaking Engagements

If an autonomous agent or a server-agent in an engagement chain *disengages* another server-agent, either through destroying the goal itself or because the agent is no longer required to achieve it, all subsequent engagements in the chain

are destroyed. This is because the subsequent agents no longer satisfy a goal that can be attributed to an autonomous agent. Thus, whenever an engagement is broken, all subsequent engagements of that engagement chain are also broken.

The schema, *Disengage*, formally defines the breaking of a direct engagement between engaging and engaged agents, *engaging*? and *engaged*? respectively, and specifies the propagation of broken direct engagements for the goal, *goal*?, down through the associated engagement chain, *chain*?. All of these components are inputs. The predicates ensure that there is a direct engagement between *engaging*? and *engaged*? with respect to *goal*?, that *chain*? is an engagement chain, and that the goal of the chain is equal to the input *goal*?. The set of cooperations remains unchanged, but the engagement chain, *chain*?, is removed from the system and replaced with the chain resulting from cutting the original chain at the engaging agent. Finally, the direct engagements are updated accordingly.

Disengage

$$
\begin{array}{l}
engaging? : Agent \\
engaged? : ServerAgent \\
goal? : Goal \\
chain? : EngagementChain \\
\Delta MultiAgentSystemStructure \\
\hline
MakeEng(goal?, engaging?, engaged?) \in direngagements \\
chain? \in engchains \\
chain?.goal = goal? \\
cooperations' = cooperations \\
engchains' = engchains \setminus \{chain?\} \cup \{CutChain(chain?, engaging?)\} \\
direngagements' = direngagements \setminus \\
\quad \{d : DirectEngagement \mid \\
\quad\quad (\langle d.client, d.server \rangle \mathsf{in}\ chain?.agentchain) \\
\quad\quad\quad\quad \wedge\ d.goal = chain?.goal \bullet d\} \cup \\
\quad \{d : DirectEngagement \mid \\
\quad\quad (\langle d.client, d.server \rangle \mathsf{in}\ (CutChain(chain?, engaging?)).agentchain \\
\quad\quad\quad\quad \wedge\ d.goal = chain?.goal) \bullet d\}
\end{array}
$$

4.3 Joining a Cooperation

A cooperation occurs when an autonomous agent generates a goal by recognising that goal in another autonomous agent. There are two cases when one autonomous agent, C, adopts the goal, g, of another, G. If no cooperation exists between any other autonomous agents and G with respect to g, then a new cooperation structure is created. Alternatively, if a cooperation already exists between another agent and G with respect to g, then C joins this cooperation.

Formally, the following schema, *GeneralCooperate* describes the general system change when a cooperating agent, *coopagent*?, adopts the goal, *goal*?, of the generating agent, *genagent*?, where all are inputs. The predicate part of the schema states that *genagent*? has *goal*?, that *coopagent*? does not, and that both

agents are autonomous. The sets of direct engagements and engagement chains remain unchanged.

GeneralCooperate

$$
\begin{array}{l}
goal? : Goal \\
genagent?, coopagent? : AutonomousAgent \\
\Delta MultiAgentSystemStructure \\
\hline
goal? \in genagent?.goals \\
goal? \notin coopagent?.goals \\
\{genagent?, coopagent?\} \subseteq autonomousagents \\
direngagements' = direngagements \\
engchains' = engchains
\end{array}
$$

Then a new cooperation is formed when there is no existing cooperation for *goal*? with *genagent*? as the generating agent. This is described formally below.

NewCooperation

$$
\begin{array}{l}
GeneralCooperate \\
\hline
\neg\,(\exists\, c : cooperations \bullet c.goal = goal? \land c.generatingagent = genagent?) \land \\
\quad cooperations' = cooperations \cup \\
\qquad \{MakeCoop(goal?, genagent?, \{coopagent?\})\}
\end{array}
$$

If such a cooperation does exist, then *coopagent*? is added to it. Specifically, it adopts the goal, *goal*?, and joins the cooperation, as specified in the *JoinCooperation* schema.

JoinCooperation

$$
\begin{array}{l}
GeneralCooperate \\
\hline
\exists\, c : cooperations \bullet c.goal = goal? \land c.generatingagent = genagent? \land \\
\qquad c.goal = goal? \land c.generatingagent = genagent? \land \\
\quad cooperations' = cooperations \setminus \{c\} \cup \\
\qquad \{MakeCoop(goal?, genagent?, c.cooperatingagents \cup \\
\qquad\qquad \{ObjectAdoptGoals(coopagent?, \{goal?\})\})\}
\end{array}
$$

4.4 Breaking or Leaving a Cooperation

There are three cases for autonomous agents destroying the goal of a cooperation in which they are involved, illustrated in Figure 2. First, the generating agent, G, destroys the goal of a cooperation with the result that the cooperation is itself destroyed. This does not imply that $C1$ and $C2$ have destroyed the goal. Second, the cooperation is also destroyed when the only cooperating agent destroys the cooperation goal. Finally, when there are many cooperating agents, one of which destroys the cooperation goal, the cooperation is not destroyed but modified so that only one agent *leaves* the cooperation.

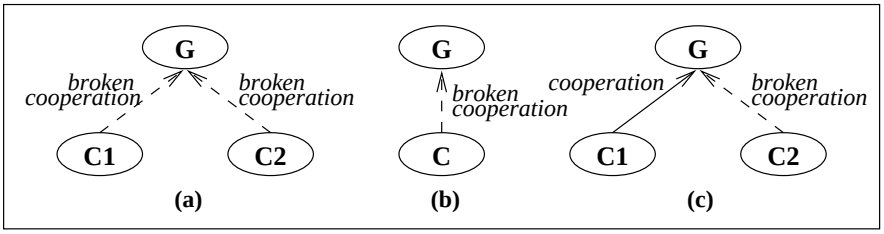

Fig. 2. Breaking a cooperation

In all three cases, however, the state is changed and the set of engagements is unaltered as defined by *CommonBreakCooperation*. A goal, *goal?*, a cooperation, *coop?*, and, optionally, two autonomous agents, *genagent?*, and *coopagent?* are inputs. The preconditions state that either *genagent?* or *coopagent?* is input. In addition, the schema checks that *genagent?* is the generating agent and that *coopagent* is a cooperating agent of *coop?*. The sets of direct engagements and engagement chains are unchanged.

CommonBreakCooperation

$$
\begin{array}{l}
goal? : Goal \\
coop? : Cooperation \\
genagent?, coopagent? : \mathsf{optional}\ [AutonomousAgent] \\
\Delta MultiAgentSystemStructure \\
\hline
\#(genagent? \cup coopagent?) = 1 \\
genagent? \subseteq \{coop?.generatingagent\} \\
coopagent? \subseteq coop?.cooperatingagents \\
direngagements' = direngagements \\
engchains' = engchains
\end{array}
$$

Each of the three different scenarios can now be specified formally as refinements to this general operation schema. First, the generating agent of the cooperation destroys the goal of the cooperation. The cooperation, *coop?*, is destroyed and removed from *cooperations*.

GeneratingAgentDestroysCooperation

$$
\begin{array}{l}
CommonBreakCooperation \\
\hline
\mathsf{defined}\ genagent? \Rightarrow \\
\quad cooperations' = cooperations \setminus \{coop?\}
\end{array}
$$

Second, the only cooperating agent destroys the goal of the cooperation. In this case, the cooperation is similarly destroyed and removed from *cooperations*.

CooperatingAgentDestroysCooperation

CommonBreakCooperation

$$(\mathsf{defined}\ coopagent? \land coopagent? = coop?.cooperatingagents) \Rightarrow cooperations' = cooperations \setminus \{coop?\}$$

Finally, a cooperating agent which is not the only cooperating agent destroys the goal of the cooperation. It is removed from the cooperation and the remaining cooperation is added to *cooperations*.

CooperatingAgentLeavesCooperation

CommonBreakCooperation

$$(\mathsf{defined}\ coopagent? \land coopagent? \subset coop?.cooperatingagents) \Rightarrow cooperations' = cooperations \setminus \{coop?\} \cup \{MakeCoop(goal?, coop?.generatingagent, (coop?.cooperatingagents \setminus coopagent?))\}$$

Schema disjunction is then used to define *BreakCooperation*.

$$BreakCooperation \mathrel{\widehat{=}} GeneratingAgentDestroysCooperation \lor CooperatingAgentDestroysCooperation \lor CooperatingAgentLeavesCooperation$$

5 Discussion

Identifying the structures and relationships between agents in multi-agent systems provides a way of understanding the nature of the system, its purpose and functionality. This is typical of existing analyses. However, if we are to build systems based on a recognition of these relationships, so that prior relationships are not destroyed when new ones are created, we must extend such analyses into *operational* areas. This paper has provided just such an analysis, explicating the different kinds of structures that can arise, and showing how different configurations of agents and relationships can evolve by invoking and and destroying them. As part of the analysis, we have had to consider a range of scenarios and the effects of changing relationships upon them.

This is particularly important for system developers aiming to build programs that are capable of exploiting a dynamic multi-agent environment. An operational analysis is vital in providing a link from the structural account to methods for accessing and manipulating these structures. If such operational analyses are avoided, then the merit of research that aims to lay a foundation for the practical aspects of multi-agent systems is limited. Indeed, most existing research has concentrated on addressing either theoretical or practical problems, and has not managed to cross the boundary between the two. Our work strives to place a foot firmly in both camps. We have constructed a formal specification of the structures necessary for an understanding of multi-agent systems and the relationships therein, and we have also shown how it is possible to set about using this understanding in the development of practical systems. Certainly, more

work is required for a complete move from theory to practice, but the current work takes us a large part of the way down that road.

References

1. M. d'Inverno and M. Luck. A formal view of social dependence networks. In *Proceedings of the First Australian DAI Workshop, Lecture Notes in Artificial Intelligence, 1087*, pages 115–129. Springer Verlag, 1996.
2. E. H. Durfee and T. Montgomery. MICE: A flexible testbed for intelligent coordination experiments. In *Proceedings of the 1989 International Workshop on Distributed Artificial Intelligence (IWDAI-89)*, 1989.
3. B. A. Huberman and S. H. Clearwater. A multiagent system for controlling building environments. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 171–176, San Francisco, CA, June 1995.
4. V. Lesser. Preface. In *Proceedings of the First International Conference on Multi-Agent Systems*, page xvii, 1995.
5. M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.
6. M. Luck and M. d'Inverno. Goal generation and adoption in hierarchical agent models. In *AI95: Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*. World Scientific, 1995.
7. M. Luck and M. d'Inverno. Engagement and cooperation in motivated agent modelling. In *Proceedings of the First Australian DAI Workshop, Lecture Notes in Artificial Intelligence, 1087*, pages 70–84. Springer Verlag, 1996.
8. J. M. Spivey. *The Z Notation*. Prentice Hall, Hemel Hempstead, 2nd edition, 1992.

Appendix: Z Extensions

We have found it useful in this specification to be able to assert that an element is optional. The following definitions provide for a new type, optional T, for any existing type, T, along with the predicates defined and undefined which test whether an element of optional T is defined or not. The function, the, extracts the element from a defined member of optional T.

$$\mathsf{optional}\,[X] == \{xs : \mathbb{P}\, X \mid \#xs \leq 1\}$$

$$\begin{array}{|l}
\hline
[X] \\
\mathsf{defined}_, \mathsf{undefined}_ : \mathbb{P}(\mathsf{optional}\,[X]) \\
\mathsf{the}: \mathsf{optional}\,[X] \pfun X \\
\hline
\forall\, xs : \mathsf{optional}\,[X] \bullet \mathsf{defined}\ xs \Leftrightarrow \#xs = 1 \land \\
\qquad\qquad \mathsf{undefined}\ xs \Leftrightarrow \#xs = 0 \\
\forall\, xs : \mathsf{optional}\,[X] \mid \mathsf{defined}\ xs \bullet \\
\qquad\qquad \mathsf{the}\ xs = (\mu\, x : X \mid x \in xs) \\
\hline
\end{array}$$

This article was processed using the LaTeX macro package with LLNCS style