Flexible Agent Grouping in Executable Temporal Logic

Michael Fisher and Tony Kakoudakis
Department of Computing and Mathematics
Manchester Metropolitan University
Manchester M1 5GD, U.K.
{M.Fisher,T.Kakoudakis}@doc.mmu.ac.uk
http://www.doc.mmu.ac.uk/RESEARCH/LoCo/

Abstract

In recent years, it has become apparent that traditional software techniques are unable to cope with the complexity of today’s requirements and so a new technology, termed agent-based systems, has been developed and applied. While research has been carried out concerning the capabilities and design of individual autonomous agents, complex real-world applications often require the cooperation and teamwork of a group of agents. In general a single agent suffices as long as the agent has the resources and the capacity to achieve the required goals. However, as the complexity and uncertainty of the tasks involved in the achievement of the system’s goals increases, cooperative work between a number of agents becomes essential. In this case, a mechanism that groups and organises the cooperative activity of individual agents is required.

1 Introduction

Concurrent METATEM [6] is an agent-based programming language comprising two elements: the representation of each individual agent’s behaviour using a temporal specification; and an operational framework for agents providing both asynchronous concurrency and broadcast message-passing. The language we use for describing individual agent behaviour is based on discrete linear temporal logic [4]. This provides the expressive power necessary for describing the dynamic activity of an agent, yet is still simple enough to be used as the basis for a programming language for agent-based systems [7]. As such it provides a coherent and consistent programming model within which a variety of agent applications can be represented.

In previous papers, for example [7], we have suggested that agent grouping mechanisms [19] could be of great importance within the Concurrent METATEM language. In particular, by using such an approach, the agent space can be structured so that an agent’s messages are broadcast only to other agents within the same group. More recently, we have advocated the modification of the operational properties existing within groups (e.g. communication properties, synchrony/asynchrony) [12]. However, the simple group structuring mechanisms that have been implemented for Concurrent METATEM, for example in [16], remain limited in their scope.
In this paper we take a new look at grouping in Concurrent METATEM. Starting just with the basic language, we propose a powerful and flexible grouping mechanism that is conceptually simple, and that will fit in with the use of executable temporal logic. Thus, we will tackle the following questions:

- Exactly what is a group in Concurrent METATEM and how does it relate to other agents?
- How do we use temporal logic to specify (and implement) the dynamic properties of groups?

2 The Computational Model

The language we use for describing agent behaviour is Concurrent METATEM [6]. This not only provides a notation for formally representing agents, but can be seen as a programming language in that the logical representations can be directly executed [9]. The basic tenets of this approach are that:

1. everything is an agent,
2. all agents are concurrently active, and,
3. the basic mechanism for communication between agents is broadcast message-passing.

2.1 Agents

Agents, encapsulating both data and behaviour, are the basic entities within our model. Individual agents only act upon certain identified messages received from their environment. Thus, an agent must be able to filter out messages that it wishes to recognise, ignoring all others. The definition of which messages an agent recognises, together with a definition of the messages that an agent may itself produce, is provided by the interface definition for that particular agent. For example, the interface definition for a ‘car’ agent may be defined in the following way:

```plaintext
car()
in: go, stop, turn
out: fuel, overheat
```

Here, \{go, stop, turn\} represents the set of messages the agent is able to recognise, while \{fuel, overheat\} represents the set of messages the agent itself is able to produce.

2.2 Concurrency and Communication

It is fundamental to our approach that all agents are (potentially) concurrently active; by default they are asynchronously executing.

The basic communication mechanism used between agents is broadcast message-passing. Thus, when an agent sends a message it does not necessarily send it to a specified destination, it merely sends it to its environment, where it can potentially be received by all other agents. Although broadcast is the basic communication mechanism, both multicast
and point-to-point message-passing (achieved by adding an extra ‘destination’ argument to the message) can be defined in terms of this.

The default communication behaviour is that if a message is broadcast, then it will eventually be received at all possible receivers. In semantic terms, this can be formalised as:\footnote{Note that ‘$\Diamond$’ is the temporal operator representing “at some time in the future.” See Section 2.3.}

\[
\text{broadcast}(\text{msg}) \Rightarrow \forall a \in \text{Agents}. \Diamond \text{receive}(a, \text{msg})
\]

### 2.3 Temporal Logic

The language we use for describing individual agent behaviour is based on temporal logic [4]. The propositional temporal logic we use is based on a linear, discrete model of time [14]. Thus, time is modelled as an infinite sequence of discrete states, with an identified starting point, called “the beginning of time”. Classical formulae are used to represent constraints within states, while temporal formulae represent constraints between states. This temporal logic can be seen as classical logic extended with various modalities, for example ‘$\Diamond$’, ‘$\square$’, and ‘$\bigcirc$’. The intuitive meaning of these connectives is as follows: $\Diamond A$ is true now if $A$ is true sometime in the future; $\square A$ is true now if $A$ is true always in the future; and $\bigcirc A$ is true now if $A$ is true at the next moment in time. Similar connectives can also be introduced to enable reasoning about the past [18].

### 2.4 Temporal Execution

As an agent’s behaviour is represented by a temporal formula, this can be transformed into Separated Normal Form (SNF) [10], which not only removes the majority of the temporal operators, but also translates the formula into a set of rules suitable for direct execution. Each of these rules is of one of the forms presented in Figure 1, where ‘start’ means “at the beginning of time (i.e. execution)”, and each $k_i$ or $m_j$ is a literal. This normal form provides a simple structure for formal temporal descriptions of agents.

![Figure 1: Separated Normal Form.](image)

In order to animate the behaviour of an agent, we choose to execute its temporal specification directly [9]. Execution of a temporal formula corresponds to the construction of a model for that formula and, in order to execute a set of SNF rules representing the behaviour of an agent, we utilise the imperative future [2] approach. This evaluates the SNF rules...
rules at every moment in time, using information about the history of the agent in order to constrain future execution. Thus, a forward-chaining process is employed to produce a model for a formula; the underlying (sequential) METATEM language [1] exactly follows this approach.

The operator used to represent the basic temporal indeterminacy within the SNF rules is the sometime operator, ‘◊’. When ◊φ is executed, the system must try to ensure that φ eventually becomes true. As such eventualities might not be able to be satisfied immediately, we must keep a record of the unsatisfied eventualities, retrying them as execution proceeds.

As an example of a simple set of rules which form a fragment of an agent’s description, consider the following which could be rules forming part of the behaviour of the car agent.

\[
\begin{align*}
  \text{start} & \Rightarrow \text{moving} \\
  \text{stop} & \Rightarrow ◊\neg\text{moving} \\
  (\text{fast} \land \text{go}) & \Rightarrow O(\text{overheat} \lor \text{fuel})
\end{align*}
\]

Here, we see that moving is true at the start of execution. Note that, as start can only be satisfied at the first moment in time, the first rule has no effect beyond the first state of computation. Whenever stop is satisfied, a commitment to eventually make moving false is made. Similarly, whenever both go and fast are satisfied, then either overheat or fuel must be made true in the next moment in time.

While we will not consider the execution mechanism in detail, we note that:

1. clearly there is indeterminacy via both the execution of disjunctions and of ‘◊’ operators — the execution can backtrack and it can be shown that, under certain heuristics, completeness will be retained [2];

2. while we have introduced propositional temporal logic in this section, we will also use elements from first-order temporal logic, particularly in more complex examples. Details of the first-order framework can be found in [6].

3 Requirements of a Grouping Mechanism

Informally, a group is a structuring mechanism that collects together and organises a number of individual agents. This mechanism restricts a multi-agent system’s communication patterns such that a particular agent in the system can only send messages within the boundaries of the groups that it is a member of. Thus, the agents in a group will have the ability to restrict communication so that it remains local to the group and thus contributes only to the achievement of their goals. Having such a mechanism available, system designers will have the ability to construct flexible, reliable and diverse agent-based systems as well as the ability to structure and organise the (potentially) chaotic agent space.

Before providing a model for groups (and agents), let us consider the capabilities we require of groups in general. The grouping mechanism should:

1. allow complex structuring of the agent space, for example in order to construct patterns of agents (organisations) conforming to a particular problem,

2. allow dynamic creation of groups and dynamic movement of agents to and from groups,
3. permit the modification of environmental conditions within a group,
4. have a precise formal semantics, and,
5. allow groups and agents to be treated in a similar way, thus avoiding the need to introduce two separate mechanisms.

Perhaps more important than all of these though is the requirement that the group mechanism be simple and intuitive!

4 A Model for Groups

The key observation we make is that groups should be treated in a similar way to agents in Concurrent METATEM. In particular, groups and agents are basically the same entities. Since the group is also an agent, it still has the basic elements of agency within Concurrent METATEM, notably

- groups have interfaces — thus, in contrast to simpler forms of grouping, these groups can both filter out incoming messages before they reach agents within the group and send other messages that none of the agents inside the group could have sent,
- groups contain rules describing how to deal with incoming messages,
- groups can send out multiple messages at every step, and
- groups can clone themselves.

For example, a group might contain the agents $a_1$, $a_2$, $g_3$, and $g_4$ (where $g_3$ and $g_4$ are themselves groups), and might have environmental conditions of instantaneous broadcast and synchronous execution.

Thus, the core components of an agent can be defined as

\[
\text{Agent} ::= \text{Behaviour} : \text{TLP Spec} \\
\quad \text{Contents} : \mathcal{P}(\text{Agent}) \\
\quad \text{Context} : \mathcal{P}(\text{Agent})
\]

rather than just a temporal specification, as in [8]. These three main components define

- its behaviour, i.e. a temporal logic specification,
- its contents, i.e. the set of agents located within this agent, and
- its context, i.e. the set of agents in which this agent is located.

Now consider some instances of this general definition and see what they correspond to.

1. A simple agent would not contain any sub-agents,
   i.e. $\text{Contents} = \emptyset$.

2. A basic group contains other agents, but does not have any specific behaviour of its own (other than passing on messages when required),
   i.e. $\text{Behaviour} = \emptyset$. 

145
3. A more complex group, such as a ‘team’ [3] would not only contain agents, but would provide additional behaviour concerning the group,

\[ \text{i.e., } \text{Contents} \neq \emptyset \text{ and Behaviour} \neq \emptyset \]

Thus, while a simple agent just contains temporal rules (providing its behaviour), a group agent contains a set of other agents together with temporal rules defining the behaviour of the group as a whole.

4. A ‘top-level’ agent/group would not occur within any group context,

\[ \text{i.e., } \text{Context} = \emptyset \]

The complex variety of agent considered in (3) above, often characterised as teams or organisations [15], captures the notion that the group has additional properties, over and above just the containment of other agents. It is in the Behaviour specification that these properties can be defined and they could, in principle, be of a variety of types, e.g.

- constraints on execution within the group, e.g. synchronous/asynchronous [12];
- constraints on communication that can take place within the group, e.g. instantaneous, eventually, ‘lossy’ [12];
- constraints on common attitudes within the group, e.g. joint goals/intentions [17].

Below we give examples of such properties, represented as temporal formulae. Note that, in a sense, these can be seen as characterising the underlying computational properties of the group.

- Passing messages on to group members. When an agent receives a message, it may choose to pass this message on to its contents:

\[ rcv(self, \text{Msg}) \Rightarrow \forall A \in \text{Contents. send(A, Msg)} \]

- Passing messages up to containing groups. When an agent receives a message, it may choose to pass this message on to its context:

\[ rcv(self, \text{Msg}) \Rightarrow \forall A \in \text{Context. send(A, Msg)} \]

- Communication properties. For example, a message sent within a group will eventually be received by the group members:

\[ \text{send(A, Msg)} \Rightarrow \Diamond rcv(A, \text{Msg}) \]

- Communication properties. For example, a message sent within a group will be received by the group members at the next moment in time:

\[ \text{send(A, Msg)} \Rightarrow \bigcirc rcv(A, \text{Msg}) \]

- Communication properties. For example, a message sent within a group may be lost:

\[ \text{send(A, Msg)} \Rightarrow (\Diamond rcv(A, \text{Msg}) \lor \text{lost(Msg)}) \]

- Execution properties, e.g. synchrony:
\[\text{step } \Rightarrow \forall A \in \text{Contents}. \bigcirc \text{send}(A, \text{step})\]

- Execution properties, e.g. asynchrony:
  \[\text{step } \Rightarrow \forall A \in \text{Contents}. \Diamond \text{send}(A, \text{step})\]

Note that the rules characterising the behaviour of \text{rcv}, \text{step}, etc., would occur in the member agents themselves, e.g.
\[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow \text{Msg}\]

5 Implementing Group Properties

While rules of the form given above provides a meta-level view of group properties, a ‘real’ implementation would provide these behaviours directly. However, just as the difference between ‘compiled-in’ behaviours and meta-circular interpreters in declarative languages, the above approach allows us to dynamically modify the properties of groups. Similarly, finer control, such as requiring that only one message is consumed from the input queue at every moment in time, can be specified in this way.

To implement the above specifications, we might again choose to directly execute them. In principle, all we have to do is to provide such specifications and rules within the \text{Behaviour} part of the agent representing the group. Once we have access to the agent’s \text{Contents} and \text{Context} sets, our current proposal\(^2\) is just to add a little syntactic ‘sugar’ to allow \text{METATEM} rules to quantify over these sets, as follows.

1. \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow \forall A \in \text{Contents}. \text{send}(A, \text{Msg})\]
   becomes
   \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow [\text{send}(A, \text{Msg})] \text{all.of Contents}\]

2. \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow \exists A \in \text{Contents}. \text{send}(A, \text{Msg})\]
   becomes
   \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow [\text{send}(A, \text{Msg})] \text{some.of Contents}\]

3. \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow \exists! A \in \text{Contents}. \text{send}(A, \text{Msg})\]
   becomes
   \[\text{rcv}(\text{self}, \text{Msg}) \Rightarrow [\text{send}(A, \text{Msg})] \text{one.of Contents}\]

Note that the variables on the left-hand sides of rules are bound by the time the right-hand side is evaluated. Thus, keywords such as \text{some.of} and \text{all.of} bind the remaining unbound variable.

Using this type of syntax we can see how to send messages ‘upward’ to agents that contain the current one (via the \text{Context} set), ‘downwards’ to agents that the current one contains (via the \text{Contents} set) and to other agents within the same groups (the default behaviour).

\(^2\)Note that this approach has not yet been fully implemented.
6 Manipulating Groups

6.1 Adding and Removing Members

How, then, can we add to, and delete from, groups? Simply by broadcasting messages. For example the following might be the sort of messages that can be broadcast.

\[\text{add}(\text{Agent}, \text{Group}, \text{Auth})\] — once the Group agent receives this, it can choose to add Agent to its Contents, possibly dependent upon the authorisation, Auth. If Agent is added, the Group agent might broadcast \(\text{added}(\text{Agent}, \text{Group})\).

\[\text{remove}(\text{Agent}, \text{Group}, \text{Auth})\] — once the Group receives this, it might remove Agent from its Contents, again dependent upon the authorisation, Auth. If successful, Group might broadcast \(\text{removed}(\text{Agent}, \text{Group})\).

\[\text{member?}(\text{Agent}, \text{Group})\] — here the Group checks to see if Agent occurs in its Contents and can then potentially broadcast the required information, i.e. either \(\text{is\_member}(\text{Agent}, \text{Group})\) or \(\text{not\_member}(\text{Agent}, \text{Group})\).

Note two things concerning these messages:

1. all other agents (within the same group) can see these messages as they are broadcast, and,
2. the Group agent might choose not to carry out the effects of these messages — i.e. as groups evolve, they might generate their own criteria for addition/removal!

While the description of the messages above may seem vague, the key point is that it is the individual agent’s/group’s rules of behaviour that define how, if at all, these messages are treated.

6.2 Creating Groups

Just as the basic way to make new agents is to clone and existing agent, we make new groups by cloning. However, as agents and groups are the same entities, then we just clone a new agent and start adding members to it. Note that all agents have the basic capabilities of being able to answer messages concerning adding and removing agents. Thus, a wide range of complex grouping structures can be generated just via cloning and adding and removing agents. For example, once we create a clone of a group, we get a new group with the same Contents and Context. This second group can then add or remove agents without affecting the structure of the original group.

7 Conclusions

We have introduced a flexible notion of grouping simply by extending the definition of an agent so that a simple agent appears as a special case of a general group. Given this close link, it is not surprising that, as agents can have temporal behaviour, groups can have temporal behaviour. In addition, an agent can be given contents and, once it does, it too can be considered as a group. This uniformity between agents and groups allows many useful attributes, such as cloning and communication, to be directly available to groups.
Our current work concerns implementation and refinement of the syntax to be used. In particular, we are looking at how to manage consistency between contents and contexts in a continuously evolving system. Also, we are investigating the most appropriate way to represent and implement group properties, as outlined in §5.

The semantics of METATEM must be modified in order to cope with such grouping. However, we expect this to be relatively easy since the temporal specification within the group provides the group’s behaviour.

Our future work will involve adapting this flexible grouping mechanism to more powerful versions of METATEM, notably the extensions to handle modalities such as (bounded) belief [13] and goals [11]. This will allow us to tackle the representation and implementation of teams and organisations [5], as well as simpler grouping structures specified via temporal logics.

Finally, we intend to use a dynamic grouping mechanism of the type outlined in this paper to represent and simulate more complex scenarios, such as cooperation and competition, amongst societies of agents.

References


