Journal of Applied Logic 9 (2011) 223–238

Contents lists available at SciVerse ScienceDirect

Journal of Applied Logic

www.elsevier.com/locate/jal

Agent deliberation in an executable temporal framework

Michael Fisher

Department of Computer Science, University of Liverpool, United Kingdom

ARTICLE INFO

Article history:
Available online 21 September 2011

Keywords:
Agents
Executable logics
Deliberation

ABSTRACT

Autonomous agents are not so difficult to construct. Constructing autonomous agents that will work as required is *much* harder. A clear way in which we can design and analyse autonomous systems so that we can be more confident that their behaviour is as required is to use formal methods. These can, in principle, allow us to exactly *specify* the behaviour of the agent, and *verify* that any implementation has the properties required. In addition to using a more formal approach, it is clear that problems of conceptualisation and analysis can be aided by the use of an appropriate abstraction.

In this article we tackle one particular aspect of formal methods for agent-based systems, namely the formal representation and implementation of *deliberation* within agents. The key aspect here is simplicity. Agents are specified using a relatively simple temporal logic and are executed by directly interpreting such temporal formulae. Deliberation is captured by modifying the way in which execution handles its temporal goals. Thus, in this article we provide motivations, theoretical underpinnings, implementation details, correctness arguments, and comparisons with related work.

1. Introduction

The rapid growth of both the INTERNET and Grid Computing, together with the inappropriateness of traditional approaches to developing complex, distributed applications, has led to the increasingly widespread adoption of new, *multi-agent* solutions [57,60]. The abstraction central to such approaches is that of an *agent*, which is typically an autonomous software component, communicating and cooperating with other agents in order to achieve common goals [62]. This technology has been particularly successful in producing distributed systems where centralised control is either impractical or undesirable. Not only is the ability of agents to act autonomously vital, but such agents are often required to dynamically adapt to unforeseen circumstances and to work cooperatively with other agents in order to overcome problems. In this sense, such agents are truly *autonomous*, being responsible for *deliberating* over a range of possibilities and for deciding on their own course of action. Consequently, agent technology has been applied in a wide variety of areas, from industrial process control to cooperative information retrieval [32,1].

While considerable research has been carried out concerning the development of theories of agency, negotiation and cooperation, notably BDI [48] and KARO [40], there are few high-level languages for representing such agents. Although some alternatives are described in Section 6.1, most agent systems are developed directly in Java. While this is a general purpose solution, and indeed agent *shells* have been developed for Java [58,31,3], this approach is often characterised by the absence of a clear *semantics* for the agent activity, and a lack of *clarity* concerning exactly what deliberative aspects are involved.

This lack of appropriate high-level, semantically clear, agent programming languages often means that implemented systems have very little connection with high-level agent theories, such as BDI [48], though there are notable attempts

E-mail address: MFisher@liverpool.ac.uk.

1570-8683/$ – see front matter
doi:10.1016/j.jal.2011.09.004

in this direction, for example [51,44,36,14,63,6,12]. However, while agent-oriented software engineering is advancing [61], there is, as yet, little evidence of a *formal* engineering approach to the development of agent-based systems that can truly bridge the gap between theory and implementation [8]; our work is a further step in this direction. In essence, since traditional programming languages typically lack the flexibility to handle, clearly and concisely, high-level deliberation such as an agent's dynamic control of its own behaviour, we aim to provide a high-level language supporting the principled development of such agent-based systems, from logical theory to implemented system.

1.1. Content

In this article, we describe one particular approach to the formalisation and implementation of agent-based systems. Our aim here is to use high-level logical specifications for agents, then *directly execute* such specifications. This work is part of the programme of research developing the *Imperative Future* paradigm [2] for use in agent-based systems [17]. In particular, we here consolidate and extend the work originally presented in [19] on representing and implementing *deliberative* aspects, describing:

1. the use of a simple temporal logic to provide agent behaviours, including both reactive and deliberative elements;
2. an algorithm (implemented in Prolog) for executing such temporal descriptions;
3. the representation of goals by temporal eventualities and the use of orderings between goals to capture rational deliberation;
4. the implementation of such orderings both within the underlying Prolog and at the meta-level;
5. correctness arguments for the execution of deliberation via goal ordering; and
6. a range of examples, indicating how such deliberation can be utilised.

The core aspect of our work can be summarised as follows. Within temporal logic [15], the '$\Diamond$' operator represents the concept of "at some moment in the future". Thus, a formula such as '$\Diamond$ *paper_completed*' represents the constraint that, at some moment in the future the paper will be completed. While such a formula gives a declarative description of dynamic behaviour, the *Imperative Future* paradigm provides a way of directly executing such formulae to actually ensure that we *do* make *paper_completed* true at some time in the future. In this sense, we can see simple formulae containing '$\Diamond$' as representing *goals* that must be achieved. Given such descriptions, we can utilise an execution mechanism for directly executing logical formulae. The core execution mechanism actually maintains a list outstanding goals, or *eventualities*, such as *paper_completed*, and endeavours to make each one true as soon as possible. In doing so, those goals at the head of the list are attempted first. Now, by providing a mechanism for dynamically *re-ordering* such a goal list, we essentially have a way of changing the order in which the agent attempts to achieve goals. In this sense, such a re-ordering implements agent *deliberation*; a full description of this approach is given in this article.

As mentioned above, we here extend, combine and consolidate work on executable agent specifications [17] and deliberative agent specifications [19,30]. As such, this work contributes to research into *agent theory*, by providing a logical basis for representing deliberative agents, *programming language design*, providing an intuitive language for implementing deliberative agents, and *software engineering*, by providing a framework of developing agent-based systems based on executable specifications.

1.2. Structure

The structure of this article is as follows. We begin, in Section 2, by describing our view of agents and multi-agent systems, and present our desiderata for a description language for agent-based systems. Such a language is developed, in Section 3, for describing agents using temporal logic and implementing these agents via direct execution. In Section 4, we address the problem of representing and implementing deliberation between goals in a high-level manner, principally by introducing the concept of priority functions for dynamically re-ordering goals. During this, we not only contrast our executable descriptions with the standard BDI model of rational agency (Section 4.3) but also provide correctness arguments (Section 4.5). In Section 5, implementation aspects are presented, as are a number of examples. Finally, in Section 6 we provide concluding remarks and outline related and future work.

2. Agent-based systems

There is widespread use of the term 'agent', ranging from being identical to the concept of 'object' to being "an encapsulated entity with 'traditional' AI capabilities" [62]. So variable is its usage that, in some areas, the term itself is seen as being meaningless. The view we take here is that, while the key aspect of an object is *encapsulation* of state and (some) behaviour, agents are truly *autonomous*. Thus, an agent not only has control over its own state, but also can dynamically change its patterns of behaviour and communication as execution progresses (for example by 'learning' or 'forgetting'), and so can choose what form of interactions it presents to its environment. In this sense, the agent abstraction captures the core elements of autonomous systems.

We are particularly concerned with what are typically termed *rational agents* [9,10,47,59]. The key aspect of this type of agent is that, since it is autonomous, it has some *motivation* for acting in the way it does. The decisions the rational agent makes, based on these dynamic motivations, should be both 'reasonable' and 'justifiable'. Just as the use of agents is now seen as an essential tool in representing, understanding and implementing complex software systems, so the characterisation of complex components as rational agents allows the system designer to work at a much higher level of abstraction. Since we are here concerned with deliberative aspects, we will term the rational agents we examine simply as *deliberative* agents.

Agents of the above form, autonomously (and asynchronously) executing, reside in an environment consisting of other agents. The only interaction between such agents occurs through message-passing and, through this simple communication mechanism, agents can be organised into a variety of structures. As it is often the case that agents must work together, these structures typically support cooperative activity. While we will not address such dynamic organisations of agents in this article, we note that they can take many different forms, such as groups [38,22], teams [34,53,43,35,23] and organisations [25,16,64,56].

2.1. Requirements for agent description languages

In representing the internal behaviour of an individual agent, we argue that a notation satisfying most, if not all, of the following criteria is required.

- It should be high-level, yet concise, consisting of a small range of powerful constructs.
- It should possess a semantics that is both intuitive and, if possible, obvious from the syntax of the language.
- It should be able to represent not only the static, but the *dynamic*, behaviour of agents.
- It should impose as few operational constraints upon the system designer as possible (for example, concurrent activities within a single agent should be allowable and agents should be able to reside in an open, asynchronously executing, environment).

In representing an individual agent's behaviour, we choose to utilise a *formal logic*. One of the advantages of following such an approach is that the notation has a well-defined, and usually well understood, semantics. The use of a formal logic language also allows us to narrow the gap between the agent descriptions and agent theory in that the semantics of an agent is close to that of its logical description. This allows for the possibility of employing both specification and verification techniques based upon formal logic in the development of agent-based systems.

As we are considering *rational* agents, we impose additional requirements for describing, and reasoning about, rational behaviour at a high level. We note that the predominant rational agent theories all share similar elements, in particular

- an *informational* component, such as being able to represent an agent's beliefs or knowledge,
- a *dynamic* component, allowing the representation of dynamic activity, and
- a *motivational* component, often representing the agents desires, intentions or goals.

For the deliberative agents we are concerned with, we will omit the first of these, but note that this informational component is usually formalised using logics of knowledge or belief [41,46,55].

The remaining aspects are typically represented logically by temporal or dynamic logics (dynamism), and modal logics of intentions, desires or wishes (motivation). Thus, the predominant approaches to rational agent theory use relevant combinations, for example the BDI model [45] uses branching-time temporal logic (CTL*) combined with modal logics of desire (KD) and intention (KD), while the KARO framework [40] uses dynamic logic (PDL) combined with a modal logic of wishes (KD).

Unfortunately, many of these combinations become too complex (not only undecidable, but incomplete) to be used in practical situations. As we shall see later, our framework represents a simpler (and more tractable) logical basis for many aspects of deliberative agents.

3. Temporal representation and execution

In this section we will describe the basic temporal framework, which is an adaption of our earlier work on executable temporal logics [2]. The aspects particular to the representation of deliberative activity, as outlined in Section 2.1, will be addressed specifically in Section 4.

While a general logic-based approach satisfies many of the criteria in Section 2.1, we choose to use *temporal logic* as the basis of our formal description of agent behaviour. Temporal logic is a form of non-classical logic where a model of time provides the basis for the notation. In our case, a simple discrete, linear sequence of moments is used as the basic temporal model, with each moment in this temporal sequence being a model for classical logic. Such a temporal logic is more powerful than the corresponding classical logic, is still tractable (at least in the propositional case) and, as we shall describe, is useful for the description of dynamic behaviour in agent-based systems.

We begin with a brief introduction to (propositional, discrete, linear) temporal logic; see [15].

$$
\begin{array}{lcl}
\langle\sigma,i\rangle \models p & \Leftrightarrow & p \in s_i \quad [\text{where } p \in \mathcal{P}] \\
\langle\sigma,i\rangle \models \textbf{true} & & \langle\sigma,i\rangle \not\models \textbf{false} \\
\langle\sigma,i\rangle \models \textbf{start} & \Leftrightarrow & i = 0 \\
\langle\sigma,i\rangle \models \varphi \wedge \psi & \Leftrightarrow & \langle\sigma,i\rangle \models \varphi \text{ and } \langle\sigma,i\rangle \models \psi \\
\langle\sigma,i\rangle \models \varphi \vee \psi & \Leftrightarrow & \langle\sigma,i\rangle \models \varphi \text{ or } \langle\sigma,i\rangle \models \psi \\
\langle\sigma,i\rangle \models \neg\varphi & \Leftrightarrow & \langle\sigma,i\rangle \not\models \varphi \\
\langle\sigma,i\rangle \models \bigcirc\varphi & \Leftrightarrow & \langle\sigma,i+1\rangle \models \varphi \\
\langle\sigma,i\rangle \models \Diamond\varphi & \Leftrightarrow & \text{there exists a } k \in \mathbb{N} \text{ such that } k \geqslant i \text{ and } \langle\sigma,k\rangle \models \varphi \\
\langle\sigma,i\rangle \models \Box\varphi & \Leftrightarrow & \text{for all } j \in \mathbb{N}, \text{ if } j \geqslant i \text{ then } \langle\sigma,j\rangle \models \varphi \\
\langle\sigma,i\rangle \models \varphi\mathcal{U}\psi & \Leftrightarrow & \text{there exists a } k \in \mathbb{N}, \text{ such that } k \geqslant i \text{ and } \langle\sigma,k\rangle \models \psi \\
 & & \quad \text{and for all } j \in \mathbb{N}, \text{ if } i \leqslant j < k \text{ then } \langle\sigma,j\rangle \models \varphi \\
\langle\sigma,i\rangle \models \varphi\mathcal{W}\psi & \Leftrightarrow & \text{either } \langle\sigma,i\rangle \models \varphi\mathcal{U}\psi \text{ or } \langle\sigma,i\rangle \models \Box\varphi
\end{array}
$$

Fig. 1. Semantics of propositional discrete linear temporal logic.

3.1. Propositional temporal logic

The basic definition of each agent will be given by a temporal logic specification [37]. As the temporal logic used here is based on a linear, discrete model, time is represented as an infinite sequence of discrete 'moments', with an identified starting point, called "the beginning of time". Classical formulae are used to represent constraints within individual moments, while temporal formulae represent constraints *between* moments. Examples of temporal operators are:

$\Diamond\varphi$ is satisfied now if φ is satisfied at *some* moment in the future;
$\Box\varphi$ is satisfied now if φ is satisfied in *all* moments in the future;
$\varphi\mathcal{U}\psi$ is satisfied now if φ is satisfied from now *until* a future moment when ψ is satisfied;
$\bigcirc\varphi$ is satisfied now if φ is satisfied at the *next* moment in time;
start is only satisfied at the *beginning of time*.

Formally, formulae are constructed using the following connectives and proposition symbols.

- A set, $\mathcal{P}$, of propositional symbols.
- Nullary connectives, **true**, **false** and **start**.
- Propositional connectives, $\neg$, $\vee$, $\wedge$, and $\Rightarrow$.
- Temporal connectives, $\bigcirc$, $\Diamond$, $\Box$, $\mathcal{U}$, and $\mathcal{W}$.

The set of well-formed formulae of the logic, denoted by WFF, is inductively defined as the smallest set satisfying:

- any element of $\mathcal{P}$ is in WFF, as are **true**, **false** and **start**;
- if φ and ψ are in WFF then so are

$$\neg\varphi \quad \varphi \vee \psi \quad \varphi \wedge \psi \quad \varphi \Rightarrow \psi \quad \Diamond\varphi \quad \Box\varphi \quad \varphi\mathcal{U}\psi \quad \varphi\mathcal{W}\psi \quad \bigcirc\varphi$$

An *eventuality* is defined as a WFF of the form $\Diamond\varphi$, while a *state formula* is a WFF containing no temporal operators.

As mentioned above, the semantics of this logic is standard [26] with formulae being interpreted over structures isomorphic to the Natural Numbers, $\mathbb{N}$. Thus, a model, σ, can be characterised as a sequence of moments or *states*

$$\sigma = s_0, s_1, s_2, s_3, \ldots$$

where each state, s_i, is a set of propositions representing those satisfied in the ith moment in time. As formulae in this logic are interpreted at a particular state in the sequence (i.e. at a particular moment in time), the notation

$$\langle\sigma,i\rangle \models \varphi$$

denotes the truth (or otherwise) of formula φ in the model σ at moment $i \in \mathbb{N}$. If there is some σ such that $\langle\sigma,0\rangle \models \varphi$, then φ is said to be *satisfiable*. If $\langle\sigma,0\rangle \models \varphi$ for all models, σ, then φ is said to be *valid* and is written $\models \varphi$. Note that formulae here are interpreted at time 0; this is an alternative, but equivalent, definition to the one commonly used [15]. Given this form of interpretation, the semantics of formulae in WFF are given in Fig. 1.

3.2. A normal form for execution

As an agent's behaviour is represented by a temporal logic formula, the formula can be transformed into the temporal normal form, SNF [18]. This process not only removes the majority of the temporal operators within the logic, but also translates the formula into a set of *rules* suitable either for execution or verification. A formula translated into SNF is of the form

$$\bigwedge_{i=1}^{n} R_i$$

where each R_i is of one of the following varieties.

$$\textbf{start} \quad \Rightarrow \quad \bigvee_{j=1}^{r} m_j$$

$$\bigwedge_{i=1}^{q} k_i \quad \Rightarrow \quad \bigcirc \bigvee_{j=1}^{r} m_j$$

$$\bigwedge_{i=1}^{q} k_i \quad \Rightarrow \quad \Diamond l$$

These are termed *initial*, *step* and *eventuality* rules, respectively.

3.3. Why use temporal logic?

There are a number of reasons for using temporal logic to describe the dynamic behaviour of agents, some of which we outline below.

- The discrete linear model structure that is the basis of the logic is very intuitive, corresponding to discrete steps in an execution sequence with an identified starting state and an infinite (linear) execution.
- The logic contains the core elements for describing the behaviour of basic dynamic execution. For example, it contains three main descriptive elements: a declarative description of the current state; an imperative description of transitions that might occur between the current and the *next* states; and a description of situations that will occur at some, indeterminate, state in the future.
  Thus, using this logic, we are able to describe the behaviour of an agent now, in transition to the next moment in time and at some time in the future.
- The basic set of concepts within SNF are sufficient as more complex temporal properties can be translated into SNF [18]. Thus, a general temporal specification can be given and transformed into a set of rules of this basic form.

As we shall see in Section 4, of particular importance, both in the representation of dynamic behaviour and in the execution of such temporal formulae, is the simplicity of the logic. We will not provide a more detailed description of the temporal logic used, nor of the exact transformations used in our approach (for a more detailed description, see [18]).

3.4. Representing individual agents

The basic elements of our approach, namely agents, each comprise two elements: an *interface definition* and an *internal definition*. The definition of which messages an agent recognises, which messages an agent may itself produce, and what parameters the agent has, is provided by the interface definition, for example

```
searcher
     in:  new_search, add_resources, terminate
    out:  found, need_resources
```

Here, {`new_search`, `add_resources`, `terminate`} is the set of messages that the '`searcher`' agent recognises, while {`found`, `need_resources`} is the set of messages the agent is able to produce. We will say little more about such interface definitions in this article — they mainly come into play within multi-agent scenarios [30,20]. The key element we are concerned with is the agent's *internal definition*, which is given directly as a set of SNF formulae. As an example of a simple set of formulae which might be part of the `searcher` agent's description, consider the following.

$$\begin{aligned} \textbf{start} &\Rightarrow \neg \texttt{searching} \\ \texttt{new\_search} &\Rightarrow \Diamond \texttt{searching} \\ (\texttt{searching} \wedge \texttt{new\_search}) &\Rightarrow \bigcirc (\texttt{found} \vee \texttt{need\_resources}) \end{aligned}$$

Here, `searching` is false at the beginning of time and whenever `new_search` is true (for example, if a `new_search` message has just been received), a commitment to *eventually* make `searching` true is given. Similarly, whenever both `new_search` and `searching` are true, then either `found` or `need_resources` will be made true in the next moment in time.

3.5. Executing temporal agent descriptions

One way to ensure that agents are implemented according to their logical semantics, and also provide a clear link between the agent theory and the agent specification, is to directly execute the temporal logic specification [17]. This move towards executable logic specifications further narrows the gap between the actual implementation of the language and the theory underlying the system. In our case, execution of a formula, φ, of a logic, L, means constructing a model, $\mathcal{M}$, for φ, i.e. $\mathcal{M} \models_L \varphi$. Thus, during execution, we are attempting to construct a model for the formula corresponding to the specification (i.e. the set of SNF rules).

So, we have that temporal logic formulae (i.e. SNF rules) are used to specify agents and are directly executed in order to implement these agents. For this to be successful, we must be sure that the execution algorithm implements temporal logic formulae correctly. To do this, we use the *imperative future* paradigm [2]. Here, a *forward-chaining* process is employed, using information about both the history of the agent's execution and its current set of rules in order to constrain its future execution.

The key element in this form of execution is the *sometime* operator, '$\Diamond$', which is used to represent basic temporal indeterminacy. When a formula such as '$\Diamond\varphi$' is executed, the system must attempt to ensure that φ *eventually* becomes true. As such eventualities might not be able to be satisfied immediately, a record of the outstanding eventualities must be kept, so that they can be re-tried as execution proceeds. (As we will see later, this record is implemented as an ordered list.) The standard heuristic used is to attempt to satisfy, at each state, as many eventualities as possible, starting with the oldest outstanding eventuality [2]. A slightly more detailed execution algorithm is given below (see also [2]).

Here, the execution mechanism attempts to build a model for a set of SNF clauses comprising Initial, Step, and Eventuality subsets, as follows.

1. Make a (consistent) choice of assignments for propositional symbols in the Initial set of clauses; label this as S_0 and let $E_0 = <>$.
2. Given S_i and E_i, proceed to construct S_{i+1} and E_{i+1} as follows:
   (a) let $C = \{F \mid (P \Rightarrow \bigcirc F) \in \text{Step}$ and $S_i \models P\}$, i.e., C represents the Step constraints on S_{i+1}
   (b) let $E_{i+1} = E_i \frown \langle V \mid (Q \Rightarrow \Diamond V) \in \text{Eventuality}$ and $S_i \models Q\rangle$, i.e., E_{i+1} is the previous list of outstanding eventualities, extended with all the new eventualities generated in S_i
   (c) for each $V \in E_{i+1}$, starting at the head of the list, E_{i+1}, *if* $(V \wedge C)$ is consistent, *then* let $C = (V \wedge C)$ and remove V from E_{i+1}
   (d) choose an assignment consistent with C and label this S_{i+1}
   (e) **loop check**:
   if an eventuality within E_{i+1} has occurred identically in the previous N states *then* fail and backtrack to a previous choice point (if no previous choice points are available, terminate the execution)
   (f) go to (2)

Thus, the execution mechanism is allowed to backtrack. As the agent has a range of non-deterministic choices, it can, if it finds a contradiction, backtrack to a previous choice point and continue executing but on the basis of a different choice. It can also (see (e) above) be forced to backtrack if the same eventuality has been outstanding for N steps. The bound, N, is generated from the size of the formula being executed and this effectively ensures that, if an eventuality could have been satisfied, it could have been satisfied by this moment in time (on some execution sequence).

The execution of basic temporal specifications of the above form allows us to specify and implement a variety of dynamic behaviours. In particular, it allows us to develop both *reactive* and *planning* behaviours and, by allowing concurrent activities within an individual agent, we are able to represent behaviour that is a combination of these aspects. Thus, agents can react immediately to certain stimuli, but can be carrying out a longer term planning process in the background.

3.6. Examples

Reaction rules. An agent can contain a range of transition rules representing reactive situations, such as

$$\texttt{low\_fuel} \Rightarrow \bigcirc \texttt{alert}$$
$$\texttt{detect\_object} \Rightarrow \bigcirc \texttt{record\_object}$$

Note that a response occurs here in the next step of the agent and so a variety of immediate responses can be represented. As well as being useful for reactive architectures in Distributed AI and multi-agent systems, such rules can be used as part of more traditional applications, such as process control.

Planning rules. We are also able to represent simple planning activities, for example by

$$\texttt{problem} \Rightarrow \Diamond \texttt{plan}$$
$$\texttt{plan} \Rightarrow \bigcirc \texttt{announce\_solution}$$

which states that at some time in the future the agent will have generated a plan to solve a particular problem and, when it does, the agent will inform others than a solution has been found. An agent might construct the plan, using $\Diamond\texttt{plan}$, as above, but adding clauses constraining the production of the plan. For example, if we are to attempt to plan $\Diamond\texttt{paper\_completed}$, then we might have pre-conditions such as `text_completed` and `bibliography_completed`, and so might require

$$\neg\texttt{text\_completed} \Rightarrow \bigcirc\neg\texttt{paper\_completed}$$
$$\neg\texttt{bibliography\_completed} \Rightarrow \bigcirc\neg\texttt{paper\_completed}$$

which state that `paper_completed` cannot be achieved if either of the distinct preconditions `text_completed` and `bibliography_completed` have not been achieved. These preconditions might themselves be turned into sub-goals with

$$\neg\texttt{paper\_completed} \Rightarrow \Diamond\texttt{text\_completed}$$
$$\neg\texttt{paper\_completed} \Rightarrow \Diamond\texttt{bibliography\_completed}$$

meaning that if the paper is not yet completed, then $\Diamond\texttt{text\_completed}$ and $\Diamond\texttt{bibliography\_completed}$ are sub-goals.

Aside: *planning and reaction*. Thus, as above, we can attempt to utilise the deductive and backtracking aspects of the system in order to achieve the construction of a plan (a further option may be to utilise meta-level control features [2]). Note, however, that when agents are part of an open multi-agent system, backtracking past the broadcast of a message is not allowed [20]. This allows agents to carry out search through backtracking internally, but avoids the problem of attempting to rollback actions in a distributed system. Thus, once an agent has broadcast a message, it has effectively *committed* its execution to that choice. Because of this, the designer must be careful when developing systems comprising both planning and reaction aspects as the planning rules might call for search to continue, while the reaction rules might call for an immediate communication.

Again, there are a number of approaches that could be adopted here. When we require an agent that has both planning and reactive capabilities, one approach is to spawn a separate 'planning' agent which carries out the planning activity in parallel with the original agent. The original agent acts reactively to its environment, having spawned the planning agent, but once the planning agent has succeeded in producing a plan the original agent is at liberty to act upon it. An alternative approach is for the user to specify the agent in such a way that it only commits to an execution path (i.e. via broadcast communication) once the search for a solution to the planning problem has terminated. Here, agent execution is typically characterised as periods of internal (backtracking) computation, interleaved with communication events.

4. Representing and executing deliberation

We will now extend the basic execution approach to handle the key property of being able to reason about, and manipulate, goals. As remarked above, this deliberative activity is the central aspect of *rational* agents. Not only are such agents able to generate, and attempt to achieve, their own goals, they are also able to modify how subsequent goals are attempted, depending on the situation. To some extent, we have seen this in the planning examples in the previous section. It is this ability to control when, and how, goals are attempted that captures the form of deliberation we are interested in.

4.1. Deliberation examples

To motivate further the need for deliberation, let us now consider two simple examples.

Vehicle navigation. First, we examine a simple agent navigating a vehicle. The agent has

- information about the local terrain,
- information concerning target destinations, and
- motivations, such as to get to a destination, to avoid obstacles, to continue moving until a destination is reached, etc.

The agent must dynamically deliberate over (possibly conflicting) goals in order to decide what actions (for example, movement) to take and, based on its current state, generate new goals (for example to add a new destination) or revise its current goals.

Dining and wishing. As a second example, consider a 'human-like' agent that wants to *eat lunch*, *sleep* and *be famous* and so generates all these as goals. The agent keeps trying to be famous, but realises that it is does not know how to achieve this and so chooses to try one of its other goals. Thus, the agent then tries to satisfy the goal of eating lunch but realises that the sub-goal of making lunch must be achieved first. Consequently, the agent generates a goal to *make lunch* and attempts this next.

And so on. The agent must change between its goals dynamically in order to produce the required behaviour.

4.2. Deliberation via dynamic goal ordering

As we can see from the above examples, goals must be manipulated in quite flexible ways. We intend to achieve deliberation via a re-ordering of goals within the temporal execution mechanism.

Recall that, in the basic execution framework, there is a fixed strategy for implementing eventualities (for example '$\Diamond g$'), namely to attempt to satisfy the oldest outstanding eventuality first. In the earlier algorithm, this involves keeping a list of outstanding eventualities (called E_i), attempting them in order and always adding new eventualities to the end of this list. Because we want to provide significant additional flexibility in the manipulation of eventualities, we now add the possibility of re-ordering this goal/eventuality list *between* execution steps.

Since the implementation is provided within Prolog, the definition of such a goal re-ordering function is relatively simple. All we need to do is to provide a function that transforms the list of outstanding eventualities remaining from a state into another list for use in the next state. Such a function obviously takes the original list of eventualities as an argument and produces a new list of eventualities. However, the function can take many other arguments, for example the history of the execution so far, and so such functions can be very powerful. Further details and examples are provided in Section 5, but to summarise, we have a user-defined strategy/function for deciding which eventualities to attempt first/next, by the repeated use of, for example,

$$E_{i+1} = \mathit{priority\_function}(E_i, \mathit{History}).$$

4.3. Comparison with BDI deliberation

Since the BDI approach is the predominant mechanism for representing rational agents and describing deliberation within these agents, it is informative to compare our approach with the BDI one [19].

Since its inception, many real-world agent-based systems have been based upon the BDI philosophy [32], originally the PRS [27], but subsequently systems such as dMARS [33,13] and Interrap [24] and high-profile applications such as Air Traffic Control [48] and Space Probe Monitoring [42]. In addition, there are a number of extensions of Java [28] incorporating aspects of the BDI architecture [47], such as the JACK language [31]. The BDI model is by now the predominant approach to agent architectures and, indeed, is widely used for high-level control in practical autonomous systems.

The BDI model of rational agency [46] is an agent framework whereby individual rational agents are described in terms of their "mental attitudes" of *Belief*, *Desire* and *Intention* (BDI). Belief is used to represent the information state of an agent, while the other two characterise the agent's motivational state. The difference between desires and intentions is really in the way they are used — desires are longer term motivations for the agent, while intentions are really the goals the agent is currently tackling.

In BDI systems, for example the PRS [48], deliberation consists of two aspects: deciding which desires will become intentions; and deciding how to achieve those intentions. We can capture this via two functions:

$$\mathit{intentions} = \mathit{deliberate1}(\mathit{desires}, \mathit{information})$$
$$\mathit{actions} = \mathit{deliberate2}(\mathit{intentions}, \mathit{information})$$

Here, *deliberate1* effectively decides which desires to examine first (as intentions), while *deliberate2* decides which intentions to actively pursue. Both functions also examine the information the agent has about the world, its plans, previous goals, etc.

In our approach, desires and intentions are both represented by eventualities. As we will see later, we can treat eventualities differently depending on whether we see them as long-term, or immediate, goals. Thus, we see it as a very natural mechanism for representing both intentions and desires to use temporal eventualities (though Bratman [9] takes a different view). These are required to be satisfied eventually (if consistent), can be conflicting (for example $\Diamond\varphi$ and $\Diamond\neg\varphi$ is *not* inconsistent), and the execution must manipulate them in order to generate future execution.

Thus, once we choose to identify both desires and intentions as eventualities, then the *deliberate1* and *deliberate2* functions effectively work on eventualities and this idea of deliberation fits very naturally with our view of re-ordering lists of eventualities. Since the re-ordering function can be implemented by arbitrary Prolog code, the re-ordering carried out can be very flexible.

In Section 4.4 we will consider a BDI-like example in more detail.

4.4. Performing deliberation

In order to explain our approach to deliberation further, we will consider the "dining and wishing" example from Section 4.1. As well as showing how the re-ordering of lists of eventualities using a priority function can be useful, we will also

show how such priority functions can be split further, for example into re-ordering functions broadly corresponding to BDI deliberation.

Recall that, from this example, there are four goals that the agent has, captured within a list of eventualities. For a comparison with the BDI approach, let us consider these as *desires*:

$$\text{Desires} = [\Diamond \text{be\_famous}, \Diamond \text{sleep}, \Diamond \text{eat\_lunch}, \Diamond \text{make\_lunch}]$$

If we were to take the next step in the execution at this point, the eventualities would be attempted in order, beginning with $\Diamond$ be_famous. However, we wish to carry out some deliberation by re-ordering the list before we go ahead with execution. So, before proceeding to (attempt to) build the next state in the execution, we apply an ordering function. This might capture the view that we wish to ensure that the most important goal appears first and, again appealing to BDI notation, we might view these as *intentions* (remember, though, that they are all still just temporal eventualities):

$$\text{Intentions} = [\Diamond \text{be\_famous}, \Diamond \text{eat\_lunch}, \Diamond \text{sleep}, \Diamond \text{make\_lunch}]$$

Thus, the ordering function exchanged $\Diamond$ eat_lunch and $\Diamond$ sleep. We can view the ordering function as capturing deliberation from desires to intentions, characterising the agent's view of what goals are currently most important.

Given that ordering functions can use any information available in order to decide upon a new ordering, we might apply a second such function assessing what plans are available to achieve the goals. Thus, this function might re-order the list to the following list to be attempted.

$$\text{Attempt} = [\Diamond \text{make\_lunch}, \Diamond \text{eat\_lunch}, \Diamond \text{sleep}, \Diamond \text{be\_famous}]$$

Thus, here $\Diamond$ be_famous has been relegated to the end of the list since we do not have a plan capable of achieving be_famous. The next most important goal was $\Diamond$ eat_lunch. However, a precondition for achieving this is make_lunch and so $\Diamond$ make_lunch is moved to the front of the list. And so on.

In this way, the original list of outstanding eventualities has been re-ordered based on various criteria relevant to the agent. The subsequent list of eventualities is then used in the choice of the next state as normal. It is important to note that, in general, there is just one ordering function used between each step in the execution. However, in the above example we have split this into two component functions in order to emphasise the ability to capture BDI-like deliberation [19]. (Indeed, in [19], the two functions were termed the *desire priority* function and *intention priority* function, respectively.)

We will next consider correctness questions that arise with the use of ordering/priority functions.

4.5. Correctness

Correctness of the basic execution mechanism, as described in Section 3.5, is given by the following result when a set of SNF rules R is executed.

Theorem 4.1. *(See [2].) If a set of SNF rules, R, is executed using the above algorithm, then a model for R will be generated if, and only if, R is satisfiable.*

Note here that one of the crucial aspects is the proviso, given in part (2c) of the algorithm, that eventualities are attempted in order according to their position in the list E_i. The standard approach, given in [2], is that the list is ordered by age with the oldest outstanding eventuality occurring first. In the proof, this ensures that no eventuality is outstanding infinitely, yet only attempted a finite number of times.

Once the eventuality ordering mechanism is extended to include arbitrary ordering functions, as in Section 4.2, then a more general version of the above theorem is required.

Definition 4.1 *(Fair ordering strategy).* A *fair ordering strategy* is a mechanism for re-ordering lists of eventualities between each execution step that ensures that if an eventuality is outstanding for an infinite number of steps, then at some point in the execution that eventuality will continually be attempted.

Note that we here mean the *same* eventuality. If we keep on satisfying an eventuality and then re-generating it as a new goal then this is a different eventuality for the purposes of the above.

Theorem 4.2. *If a set of SNF rules, R, is executed using the above algorithm, with a fair ordering strategy at step* (2c)*, then a model for R will be generated if, and only if, R is satisfiable.*

Proof (*Outline*). Since the only real distinction between this theorem and Theorem 4.1 above is the execution of eventualities, then we simply focus on this. Consider one eventuality from E_i. Either the eventuality is satisfiable or it is not. If the eventuality is unsatisfiable execution will fail to satisfy this at any time. By the fair ordering definition above, we know

that finally the eventuality will be attempted at every step. Since the same occurs for all unsatisfiable eventualities then, at some point, loop checking will be applied and a different path will be attempted.

If the eventuality is satisfiable then either it is satisfied now or it keeps on being outstanding until (again by the fair ordering definition) it will be continually attempted. Either this leads to the satisfaction of this eventuality, or backtracking is forced and a satisfying situation will be found.

Thus, the *fair ordering strategy* restriction imposes a form of *fairness* on the choice mechanism. While this proviso effectively means that we *can* potentially explore every possibility during execution, the incorporation (in the algorithm) of a bound (N) on the number of states that eventualities can remain outstanding, together with the finite model property of the logic, ensures that all of the possible states in the model will be explored if necessary.

So, the question remains: are typical ordering functions fair ordering strategies, or is this constraint too restrictive? We should first note that the basic strategy of ordering the list of eventualities in terms of the oldest outstanding eventuality does, indeed, correspond to a fair ordering strategy.

Lemma 1. *The strategy of re-ordering the list of eventualities in terms of the oldest outstanding eventualities is a fair ordering strategy.*

Proof. Essentially this follows since, if an eventuality remains outstanding but unsatisfied, it will eventually be attempted an infinite number of times. All eventualities earlier in the list would either be satisfied (and so be removed from the head of the list) or would also remain unsatisfied infinitely. In the latter case, these earlier eventualities would not stop the eventuality in question being attempted. □

4.6. A simplified approach: using 'prefer' functions

In general, if we are to implement a function for ordering a list (for example, a list of eventualities), then we must have a predicate for comparing elements within the list. Typically, this is a version of '$<$' over the type of elements in the list. Rather than defining the ordering function explicitly, or even providing a full definition of '$<$', the approach taken in [30] is to simply define the '$<$' predicate for *selected* pairs. In this case the '$<$' predicate is actually called '*prefer*' and such preference statements occur *explicitly* within the specification rather than as part of the implementation. Thus, for the example above, we might have

$$\mathit{prefer}(\mathsf{be\_famous}, \mathsf{make\_lunch}),$$

$$\mathit{prefer}(\mathsf{be\_famous}, \mathsf{eat\_lunch}),$$

$$\mathit{prefer}(\mathsf{make\_lunch}, \mathsf{eat\_lunch}).$$

The execution mechanism then uses this information to re-order the list of eventualities at each step. This approach has a number of advantages. It is simple and direct — the specifier need not write Prolog ordering functions and, indeed, the implementation need not be in Prolog at all (the implementation used in [30] is in Java!). However, with this simplicity comes problems. Although the three *prefer* relationships above are enough to re-order the 'desires' list in our example into the final list of 'attempts', there is no guarantee that any set of *prefer* relationships will indeed produce a unique linear order. This is not so much of a problem if, as in [30], we are prepared to accept any ordering consistent with the preferences. However, what if there is *no* consistent ordering, as in the following.

$$\mathit{prefer}(\mathsf{be\_famous}, \mathsf{make\_lunch}),$$

$$\mathit{prefer}(\mathsf{eat\_lunch}, \mathsf{be\_famous}),$$

$$\mathit{prefer}(\mathsf{make\_lunch}, \mathsf{eat\_lunch}).$$

In [30], little is done about this as it is seen as the responsibility of the specifier to ensure consistency. However, in a more comprehensive approach, further analysis of the preference structures would be carried out.

For the moment, however, we will return to the explicit definition of ordering functions, and will next consider the practical implementation of various deliberation strategies, returning to the question of whether such ordering functions are fair, in the sense above.

5. Practical deliberation

So, we here discuss the practical implementation of goal ordering strategies, particularly consider whether they are fair, in the sense above.

5.1. Prolog implementation

The system is implemented in Prolog, using techniques from [2,19]. While we will not go into detail here, we just note that the algorithm from Section 3.5 is essentially the one implemented. Within this, the most interesting part for us is the re-ordering of eventualities between states. This is implemented simply by calling the following predicate.

```
priority_function(Name, History, Goals_Before, Goals_After)
```

To ensure that *something* happens, even if the user does not provide a specific priority function, a default function is provided which does not change the ordering.

```
priority_function(default, _, Goals, Goals)
```

5.2. Running example

In order to consider various ordering functions, including their definition and effects, we will just work with one simple example. This is the specification of a very minimal *planetary rover* agent, called `rover`. This has a very simple behaviour. It is able to *detect* various aspects of its environment, for example water or minerals, and is able to invoke actions to *explore* these sensed areas. The (very) basic agent description is as follows (we use a little first-order notation to improve readability).

$$
\begin{array}{rrcl}
\texttt{rover} & & & \\
\texttt{in:} & \texttt{detect} & & \\
\texttt{out:} & \texttt{explore} & & \\
\texttt{rules:} & \texttt{detect(X)} & \Rightarrow & \Diamond\,\texttt{explore(X)} \\
& \textbf{true} & \Rightarrow & \bigcirc\,(\neg\texttt{explore(mineral)} \vee \neg\texttt{explore(water)}) \\
& \textbf{start} & \Rightarrow & \neg\texttt{explore(mineral)} \vee \neg\texttt{explore(water)}
\end{array}
$$

Thus, the `rover` agent receives sensor inputs via the '`detect`' predicate and generates exploration goals accordingly. The above behaviour states that, if a sensor input is received then it is a goal of the agent to investigate that aspect at some point in the future. The second and third rules ensure that the agent cannot explore two aspects simultaneously.

In the following, we will use the same basic agent, together with the same inputs, but will describe the effect when the internal deliberation is modified (by changing the priority function). Generally, we would expect `detect` events to occur rarely but, in order to consider the deliberative behaviour of this autonomous agent *in extremis*, we ensure that such detection events occur very often.

We will simply run the above program for 8 states. In order to proceed, the execution requires a list of inputs representing sets of literals to be consumed at each execution step:

```
[ {detect(mineral), detect(waterl)}
  {detect(mineral), detect(waterl)}
  {detect(mineral), detect(waterl)}
  {detect(mineral)}
  {detect(waterl)}
  {  }
  {  }
  {  } ]
```

We now examine what happens under various priority functions (i.e. under various deliberation strategies).

5.3. Alternative deliberation strategies

Default strategy. The default ordering strategy, '`default`', leaves the order of eventualities unchanged. Thus, in executing the `rover` program with the above input, we get the following output (re-formatted for readability).

```
**State 0:     [detect(mineral), detect(water)]
  Commitments: []

**State 1:     [detect(mineral), detect(water), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 2:     [detect(mineral), detect(water), explore(water)]
```

```
  Commitments: [sometime explore(mineral)]

**State 3:     [detect(mineral), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 4:     [detect(water), explore(water)]
  Commitments: [sometime explore(mineral)]

**State 5:     [explore(mineral)]
  Commitments: [sometime explore(water)]

**State 6:     [explore(water)]
  Commitments: []

**State 7:     []
  Commitments: []
```

Here, we see the 'detect' inputs in each state which are received from the environment. In addition, we can see that various commitments, i.e. outstanding eventualities, are subsequently generated.

So, in state 0, the agent receives detect(mineral) and detect(water) and so generates goals ◊explore(mineral) and ◊explore(water) for state 1 onwards (note that predicates not mentioned in the state are assumed to be false). As there is no particular ordering on these, an arbitrary choice will be made and explore(mineral) is made true in state 1. Since explore(water) cannot now be made true, the goal ◊explore(water) remains as a commitment for the next state (i.e. it is the only element in the list of commitments). However, in state 1, detect(mineral) is again received and so ◊explore(mineral) is added to the list of eventualities to be satisfied in state 2. Since the default approach is to not re-order this list, and since the list is ordered by age (currently ◊explore(water) is ahead of ◊explore(mineral) in the list), then at state 2 ◊explore(water) is satisfied by making explore(water) true. And so on. Thus, we see that with the default strategy, there is a fair organisation of eventuality satisfaction, effectively alternating between explore(water) and explore(mineral) in cases when both need to be satisfied.

Simply fair strategy. The 'simply fair' strategy explicitly captures a minimal form of fairness based upon whether eventualities were satisfied in the previous state. Thus, the Prolog code defining the 'simply_fair' function is as follows.

```
priority_function(simply_fair, _, [], []).

priority_function(simply_fair, LastState,
                  [sometime G | Rest], [sometime G | Att]) :-
        member(G, LastState), !,
        priority_function(simply_fair, LastState, Rest, AttRest),
        append(AttRest, [sometime G], Att).

priority_function(simply_fair, LastState,
                  [sometime G | Rest], [sometime G | Att]) :-
        priority_function(simply_fair, LastState, Rest, Att).
```

Essentially, this checks if any eventuality in the list to be satisfied has already been satisfied in the previous state. If it has, then that eventuality is moved to the end of the list. Eventualities that have not been satisfied in the last state are not re-ordered by this.

Now, if we execute the above program, but this time using the 'simply_fair' priority function, we get the following output.

```
**State 0:     [detect(mineral), detect(water)]
  Commitments: []

**State 1:     [detect(mineral), detect(water), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 2:     [detect(mineral), detect(water), explore(water)]
  Commitments: [sometime explore(mineral)]

**State 3:     [detect(mineral),explore(mineral)]
  Commitments: [sometime explore(water)]
```

```
**State 4:      [detect(water), explore(water)]
  Commitments: [sometime explore(mineral)]

**State 5:      [explore(mineral)]
  Commitments: [sometime explore(water)]

**State 6:      [explore(water)]
  Commitments: []

**State 7:      []
  Commitments: []
```

Notice how this gives the same sequence of outputs as the "built in" ordering in the default example. This shows that, within Prolog, we can code a simple ordering function that does retain fairness. We also note that this ordering is a *fair ordering strategy* in the sense of Definition 4.1.

Simply unfair strategy. Finally, we define a deliberately 'unfair' strategy (called '`simply_unfair`') that prioritises the goal $\Diamond$`explore(mineral)` over the goal $\Diamond$`explore(water)`. Rather than reproducing the code again, we note that the list is re-ordered so that if $\Diamond$`explore(water)` occurs before $\Diamond$`explore(mineral)` in the list, then it is subsequently moved to after $\Diamond$`explore(mineral)`.

In running the above program using this unfair priority function, we get

```
**State 0:      [detect(mineral), detect(water)]
  Commitments: []

**State 1:      [detect(mineral), detect(water), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 2:      [detect(mineral), detect(water), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 3:      [detect(mineral), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 4:      [detect(water), explore(mineral)]
  Commitments: [sometime explore(water)]

**State 5:      [explore(water)]
  Commitments: []

**State 6:      []
  Commitments: []

**State 7:      []
  Commitments: []
```

Notice how, in the first 3 states, `explore(mineral)` was made true even though $\Diamond$`explore(water)` was outstanding. Thus, if `detect(mineral)` were received continuously, then `explore(water)` would *never* be satisfied.

Notice how this ordering strategy is *not* a fair ordering, as defined in Definition 4.1 and so correctness of the temporal execution is *not* guaranteed by Theorem 4.2.

6. Conclusions

In this article we have considered, at a foundational level, a key aspect of autonomous systems, namely the *deliberation* that occurs within individual autonomous components. In particular, we have provided a relatively simple framework for representing the goals than an agent has, and for subsequently executing these goals. Using this approach, we have shown that by *re-ordering* the list of outstanding goals, we can produce deliberative agent behaviour. We also consider correctness and implementation aspects, emphasising the simplicity, yet flexibility, of the approach. The simplicity concerns not only the logical theory, which is much less elaborate than corresponding BDI approaches, but also the implementation. The fact that the user can supply arbitrary ordering functions (in Prolog) provides a very high degree of flexibility.

We will next mention related work, followed by our future work in this area.

6.1. Related work

While a wide variety of logical theories, purporting to represent agents, have been proposed few, if any, have provided the basis for an agent programming language. Exceptions include languages such as 3APL [12], `April` [39] and AgentSpeak [44,6], and the early work on *Agent-Oriented Programming* [51,54]. Recent work on categorising and extending the variety of goals available in such languages [29] comes close to using temporal eventualities as goals as in our approach.

BDI theory [48,46] provides a popular basis for describing agent-based systems, while the BDI architecture [47] provide a model for deliberation. Although such systems have been successfully used in a number of areas, the link between implementations and the BDI agent theory is often tenuous. Consequently, the formalisation of deliberation [10] and the link from high-level specifications to their low-level realisation are required and several works on deliberation in a BDI framework have been produced, for example [11,50]. However, there are few papers combining theoretical basis, direct implementation aspects, and strong links between, such as we provide in this article.

6.2. Future work

Our long term goal with this work is the provision of a formal framework for the specification, animation and development of distributed multi-agent systems. By basing our work on a simple temporal logic, for which there are already a variety of proof methods, we have already outlined an approach to the specification and verification of multi-agent systems [21].

We have not been primarily concerned with multi-agent aspects here. However, we wish to have multiple agents of the type described in this article, asynchronously executing, and communicating via broadcast message-passing [20]. Broadcast message-passing is a natural communication mechanism to consider as it not only matches the logical view of computation that we utilise, but it is very flexible within distributed computer systems [4,7] and distributed AI [52].

As well as the notion of individual deliberating agents, we are also exploring stronger structuring mechanisms through the 'context' extension [20]. This not only restricts the extent of an object's communications, but also provides an extra mechanism for the development of strategies for organisations. In particular, this provides the basis for agent cooperation, competition and interaction.

Finally, as we move towards more expressive temporal notations, particularly first-order temporal logic, and consider multi-agent scenarios, so retaining the completeness of the execution mechanism becomes more difficult. A central part of future work is to examine such expressive extensions with respect to practical deliberation.

Acknowledgements

Over the years many excellent researchers have helped to develop the MetateM framework. Among these, particular thanks go to Chiara Ghidini, Anthony Hepple, and Benjamin Hirsch. The author would also like to thank the anonymous referees for their helpful comments.

References

[1] Agent Technology Conference, http://www.agentlink.org/agents-stockholm, 2005.
[2] H. Barringer, M. Fisher, D. Gabbay, R. Owens, M. Reynolds (Eds.), The Imperative Future: Principles of Executable Temporal Logics, Research Studies Press, 1996.
[3] F. Bellifemine, F. Bergenti, G. Caire, A. Poggi, JADE – A Java Agent Development Framework, in: Bordini et al. [5], pp. 125–147.
[4] K.P. Birman, The process group approach to reliable distributed computing, ACM Communications 36 (12) (1993) 36–53, 103.
[5] R.H. Bordini, M. Dastani, J. Dix, A. El Fallah Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, Springer, 2005.
[6] R.H. Bordini, J.F. Hübner, R. Vieira, Jason and the golden fleece of agent-oriented programming, in: Bordini et al. [5], Ch. 1, pp. 3–37.
[7] A. Borg, J. Baumbach, S. Glazer, A message system supporting fault tolerance, in: Proceedings of the Ninth ACM Symposium on Operating System Principles, New Hampshire, ACM, Oct. 1983, pp. 90–99 (in ACM Operating Systems Review, vol. 17(5)).
[8] J. Bradshaw, M. Greaves, H. Holmback, T. Karygiannis, B. Silverman, N. Suri, A. Wong, Agents for the masses? IEEE Intelligent Systems 14 (2) (1999) 53–63.
[9] M.E. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, 1987.
[10] P.R. Cohen, H.J. Levesque, Intention is choice with commitment, Artificial Intelligence 42 (2–3) (1990) 213–261.
[11] M. Dastani, F. Dignum, J.-J.C. Meyer, Autonomy and agent deliberation, in: Agents and Computational Autonomy — Potential, Risks, and Solutions, in: Lecture Notes in Computer Science, vol. 2969, Springer, 2004, pp. 114–127.
[12] M. Dastani, M.B. van Riemsdijk, J.-J.C. Meyer, Programming multi-agent systems in 3APL, in: Bordini et al. [5], Ch. 2, pp. 39–67.
[13] M. d'Inverno, M. Luck, M.P. Georgeff, D. Kinny, M.J. Wooldridge, The dMARS architecture: A specification of the distributed multi-agent reasoning system, Autonomous Agents and Multi-Agent Systems 9 (1–2) (2004) 5–53.
[14] J. Dix, S. Kraus, V.S. Subrahmanian, Temporal agent programs, Artificial Intelligence 127 (1) (2001) 87–135.
[15] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, 1990, pp. 996–1072.
[16] J. Ferber, O. Gutknecht, Aalaadin: A meta-model for the analysis and design of organizations in multi-agent systems, in: Proc. International Conference on Multi-Agent Systems (ICMAS), July 1998.
[17] M. Fisher, Representing and executing agent-based systems, in: Intelligent Agents, Proc. Workshop on Agent Theories, Architectures, and Languages, in: LNCS, vol. 890, Springer, 1995, pp. 307–323.
[18] M. Fisher, A normal form for temporal logic and its application in theorem-proving and execution, Journal of Logic and Computation 7 (4) (1997) 429–456.

[19] M. Fisher, Implementing BDI-like systems by direct execution, in: Proc. 15th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann, 1997, pp. 316–321.
[20] M. Fisher, A. Hepple, Executing logical agent specifications, in: R.H. Bordini, M. Dastani, J. Dix, A. El Fallah-Seghrouchni (Eds.), Multi-Agent Programming: Languages, Tools and Applications, Springer, 2009, pp. 1–27.
[21] M. Fisher, M. Wooldridge, Temporal reasoning in agent based systems, in: M. Fisher, D. Gabbay, L. Vila (Eds.), Handbook of Temporal Reasoning in Artificial Intelligence, in: Advances in Artificial Intelligence, vol. 1, Elsevier Publishers/North-Holland, 2005, pp. 469–496.
[22] M. Fisher, C. Ghidini, B. Hirsch, Programming groups of rational agents, in: Computational Logic in Multi-Agent Systems (CLIMA-IV), vol. 3259, Springer-Verlag, 2004, pp. 849–856.
[23] M. Fisher, C. Ghidini, T. Kakoudakis, Dynamic team formation in executable agent-based systems, in: Rouff et al. [49].
[24] K. Fischer, J. Müller, M. Pischel, A pragmatic BDI architecture, in: Intelligent Agents II, in: Lecture Notes in Artificial Intelligence, vol. 1037, Springer-Verlag, 1995, pp. 203–218.
[25] M.S. Fox, An organizational view of distributed systems, in: Distributed Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, USA, 1988, pp. 140–150.
[26] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, The temporal analysis of fairness, in: Proc. 7th ACM Symposium on the Principles of Programming Languages (POPL), ACM Press, 1980, pp. 163–173.
[27] M.P. Georgeff, F.F. Ingrand, Decision-making in an embedded reasoning system, in: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI), 1989, pp. 972–978.
[28] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison–Wesley, 1996.
[29] K. Hindriks, B.M. van Riemsdijk, Using temporal logic to integrate goals and qualitative preferences into agent programming, in: Declarative Agent Languages and Technologies VI, in: Lecture Notes in Computer Science, vol. 5397, Springer, 2009, pp. 215–232.
[30] B. Hirsch, M. Fisher, C. Ghidini, P. Busetta, Organising software in active environments, in: Proc. 5th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA), in: LNCS, vol. 3487, Springer, 2005, pp. 265–280.
[31] JACK Intelligent Agents, http://www.agent-software.com.
[32] N.R. Jennings, M. Wooldridge, Applications of agent technology, in: Agent Technology: Foundations, Applications, and Markets, Springer-Verlag, 1998.
[33] D.N. Kinny, M.P. Georgeff, Modelling and design of multi-agent systems, in: Intelligent Agents III, in: Lecture Notes in Artificial Intelligence, vol. 1365, Springer-Verlag, Heidelberg, 1996.
[34] D. Kinny, M. Ljungberg, A.S. Rao, E. Sonenberg, G. Tidhar, E. Werner, Planned team activity, in: Artificial Social Systems; Proc. 4th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-92), in: Lecture Notes in Artificial Intelligence, vol. 830, Springer, 1992, pp. 226–256.
[35] S. Kumar, P.R. Cohen, STAPLE: An agent programming language based on the joint intention theory, in: Proc. 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), IEEE Computer Society, 2004, pp. 1390–1391.
[36] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, R. Scherl, GOLOG: A logic programming language for dynamic domains, Journal of Logic Programming 31 (1–3) (1997) 59–83.
[37] Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specification, Springer, 1992.
[38] T. Maruichi, M. Ichikawa, M. Tokoro, Modelling autonomous agents and their groups, in: Y. Demazeau, J.P. Müller (Eds.), Decentralized AI 2 – Proceedings of the 2nd European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW '90), Elsevier/North-Holland, 1991.
[39] F.G. McCabe, K.L. Clark, APRIL — Agent process interaction language, in: Intelligent Agents: Theories, Architectures, and Languages, in: Lecture Notes in Computer Science, vol. 890, Springer, 1994, pp. 280–296.
[40] J.-J. Meyer, W. van der Hoek, B. van Linder, A logical approach to the dynamics of commitments, Artificial Intelligence 113 (1–2) (1999) 1–40.
[41] R.C. Moore, A formal theory of knowledge and action, in: J.F. Allen, J. Hendler, A. Tate (Eds.), Readings in Planning, Morgan Kaufmann Publishers, 1990, pp. 480–519.
[42] N. Muscettola, P.P. Nayak, B. Pell, B. Williams, Remote agent: To boldly go where no AI system has gone before, Artificial Intelligence 103 (1–2) (1998) 5–48.
[43] D. Pynadath, M. Tambe, The communicative multiagent team decision problem: Analyzing teamwork theories and models, Journal of Artificial Intelligence Research (JAIR) 16 (2002) 389–423.
[44] A. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in: Agents Breaking Away: Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW), in: LNCS, vol. 1038, Springer, 1996, pp. 42–55.
[45] A.S. Rao, Decision procedures for propositional linear-time belief–desire–intention logics, Journal of Logic and Computation 8 (3) (1998) 293–342.
[46] A.S. Rao, M.P. Georgeff, Modeling agents within a BDI-architecture, in: Proc. 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR), Morgan Kaufmann, 1991, pp. 473–484.
[47] A.S. Rao, M.P. Georgeff, An abstract architecture for rational agents, in: Proceedings of Knowledge Representation and Reasoning (KR&R-92), 1992, pp. 439–449.
[48] A.S. Rao, M. Georgeff, BDI agents from theory to practice, in: Proc. 1st International Conference on Multi-Agent Systems (ICMAS), San Francisco, USA, 1995, pp. 312–319.
[49] C. Rouff, M. Hinchey, J. Rash, W. Truszkowski, D. Gordon-Spears (Eds.), Agent Technology from a Formal Perspective, NASA Monographs in Systems and Software Engineering, Springer-Verlag, New York, USA, 2006.
[50] M.C. Schut, M. Wooldridge, S. Parsons, The theory and practice of intention reconsideration, Journal of Experimental and Theoretical Artificial Intelligence 16 (4) (2004) 261–293.
[51] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1) (1993) 51–92.
[52] R.G. Smith, A Framework for Distributed Problem Solving, UMI Research Press, 1980.
[53] M. Tambe, Teamwork in real-world dynamic environments, in: Proc. 1st International Conference on Multi-Agent Systems (ICMAS), MIT Press, 1995.
[54] S.R. Thomas, The PLACA agent programming language, in: Intelligent Agents: Theories, Architectures, and Languages, in: Lecture Notes in Artificial Intelligence, vol. 890, Springer, 1994, pp. 307–319.
[55] B. van Linder, W. van der Hoek, J.J.C. Meyer, Formalising motivational attitudes of agents, in: Intelligent Agents II, in: Lecture Notes in Artificial Intelligence, vol. 1037, Springer-Verlag, Heidelberg, Germany, 1996, pp. 17–32.
[56] J. Vázquez-Salceda, V. Dignum, F. Dignum, Organizing multiagent systems, Autonomous Agents and Multi-Agent Systems 11 (3) (2005) 307–360.
[57] G. Weiß (Ed.), Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence, MIT Press, 1999.
[58] M. Winikoff, JACK^{TM}, intelligent agents: An industrial strength platform, in: Bordini et al. [5], Ch. 7, pp. 175–193.
[59] M. Wooldridge, Reasoning about Rational Agents, MIT Press, 2000.
[60] M. Wooldridge, An Introduction to Multiagent Systems, John Wiley & Sons, 2002.
[61] M. Wooldridge, P. Ciancarini, Agent-oriented software engineering: The state of the art, in: Agent-Oriented Software Engineering, in: Lecture Notes in Artificial Intelligence, vol. 1957, Springer-Verlag, 2001.

[62] M. Wooldridge, N.R. Jennings, Intelligent agents: Theory and practice, The Knowledge Engineering Review 10 (2) (1995) 115–152.
[63] M. Wooldridge, M. Fisher, M.-P. Huget, S. Parsons, Model checking for multiagent systems: The MABLE language and its applications, International Journal of Artificial Intelligence Tools 15 (2) (2006) 195–225.
[64] F. Zambonelli, N.R. Jennings, M. Wooldridge, Organisational rules as an abstraction for the analysis and design of multi-agent systems, International Journal of Software Engineering and Knowledge Engineering 11 (3) (2001) 303–328.