Agent Reactive Component for SMART

Leonardo Mariani and Emanuela Merelli
Dipartimento di Matematica e Informatica
Università di Camerino
via Madonna delle Carceri
62032 Camerino, Italy
email:{leonardo.mariani, emanuela.merelli}@unicam.it

Abstract

The rapid growth of agent-based systems has given rise to a situation where software testing and validation of agents and multi-agent systems are difficult. The need to define an engineering methodology for the development of agent-based systems has lead to the use of a formal specification to identify and characterise the agent systems' components. In this paper, we propose to take an existing formal framework for agent systems - SMART the first framework created to unify concepts and terms for agent and autonomous agent systems - and develop it for implementation with reactive architectures. In particular, we address the agent's reactivity as a distinct and well defined process. To this end, we refine SMART to explicitly support event generation and recognition by proposing a formal description, in Z language, of each involved component. We also discuss implementation strategy for our reactive architecture, using the Macondo platform to describe the design and implementation of the reactive component as well as to provide a foundation for subsequent software testing and validation techniques.

1 Introduction

The need to retrieve information from any source at any time is the concept that will characterise a great part of future research in the computing sector. In fact, the World Wide Web seems to be the ideal place to store almost all information - information that today is surely accessible from any pc with a browser or related tools, or information that tomorrow will be provided by the support of "push" technology, where intelligent mobile agents will go out onto the web and autonomously retrieve and push the data you require to you. In this complex scenario, mobile agent systems constitute the emergent and promising field that aims to solve this problem.

A mobile agent is an entity that acts like a PRA (perceive, reason, act) cycle [Norvig and Russell, 1995]. Different interesting properties are usually associated with agents, most relevant amongst these being autonomy, reactivity, pro-activity and social ability [Wooldridge and Jennings, 1994]. An autonomous agent is an agent that acts without direct influence of a user. An agent is reactive if it responds in a timely manner to any change that occurs in the environment. An agent is pro-active if it acts in an opportunistic manner. Finally, sociological agents are agents which communicate and cooperate with other agents, and sometimes with users.

In particular, an agent can act in a unpredictable and dynamic environment where continuously, new resources can become available while others are occupied. This scenario describes a near real-time Web environment where agents act in conflict and are forced to behave opportunistically to reach a successful state. The reactive component gives the agent the ability to respond in an efficient and opportunistic manner to any change that occurs in the environment.

In the past, reaction mechanisms have been used for several goals: MARS [Cabri *et al.*, 2000] is one example of a Linda-like [Ahuja *et al.*, 1986] coordination model which uses tuple space and reaction to improve performances in multi-agent systems. Guardian [Larsson and Hayes, 1998] is an autonomous agent for medical monitoring and analysis with reactive behaviour properties. Furthermore, other agents have been created to act in complex and dynamic environments, like the real world [Liscano *et al.*, 1995].

Because several agent architectures have an explicit reactive layer (e.g. Emotional Agents described in [Camurri and Coglio, 1998]), but none of those is described in a formal and unified manner, there is no common understanding of how to implement a reactive architecture, and no formal specification to support the testing and validation of the software agent-based systems. The increased sophistication and complexity of the agent and multi-agent systems, and the related difficulties to managing the development software process, lead to serious concerns over the quality and correctness of agent software products. As addressed in [Luck and d'Inverno, 2001], formal specification can represent one of the techniques to be applied in achieving the description of an agent software design and properties in mathematical logic.

This paper addresses this issue by taking an existing formal framework for agent systems and developing it towards an implementation for reactive architectures thus providing a foundation for subsequent software testing and validation techniques.

The SMART [d'Inverno and Luck, 2001] framework taken as a sound conceptual framework is useful to precisely and unambiguously understand agent and multi-agent systems. It provides a unified framework with high-level models of agents, relationships and interactions, but which does not specify a prescribed internal architecture for agency and autonomy. It also provides a base for further development of agent architecture and agent theory in an incremental fashion through refinement and schema inclusion. In SMART, deliberative and reactive action generation is a unique process that does not allow exploitation of the advantages of the differences between actions, a limitation which, in this project, has been overcome by distinctly defining reaction from deliberative action generation. Thus, the framework has been further developed to incorporate notions of reaction, reactive agent, reactive autonomous agent. To prove the validity of the given reactivity agent specification, we chose to implement a reactive component. To that end, we needed an existing platform whose software was open-source, for which the mobility functionality was guaranteed but no reactivity was implemented. All these features we found in Macondo [Ciancarini *et al.*, 2000].

The paper is organised as follow:

In Section 2 we discuss the background of the work: SMART framework and Macondo platform. In Section 3 we describe the key agent components for the reactivity. In Section 4 we present the formal specification of the agent reactive component. In Section 5 we discuss implementation through description of the design and implementation of the reactive component and present a foundation for subsequent software testing and validation techniques for the Macondo platform. The paper ends with conclusions presented in Section 6.

2 Background

2.1 SMART

Structured and Modular Agents and Relationship Types (SMART) [d'Inverno and Luck, 2001] is a formal specification for agents and Multi Agents Systems (MAS).

In this framework the world is made up of entities that can be instantiated as objects, agents or autonomous agents. An entity consists of four main components: a set of attributes, a set of capabilities, a set of goals and a set of motivations. The formalisation of this concept in Z language [J.M.Spivey, 1992] is given in the next schema. Z schemas have two parts: the upper declarative part, which declares variables and their type, and the lower predicate part, which relates and constraints those variable.

Entity

$$attributes : \mathbb{P} Attribute$$
$$capabilities : \mathbb{P} Action$$
$$goals : \mathbb{P} Goal$$
$$motivations : \mathbb{P} Motivation$$
$$attributes \neq \{\}$$

attributes denote the perceivable qualities of an agent; *capabilities* denote actions that an agent is capable of; *goals*, as the word says, are the set of all goals for an agent, and *motivations* are the set of all goal-generation attitudes. By *Attribute* it is also possible to define the environment simplistically as follows:

$$Environment == \mathbb{P}_1 Attribute$$

An entity is an object if its set of capabilities is not empty.

Object

$$Entity$$
$$capabilities \neq \{\}$$

By refining an object, an Agent and Autonomous Agents may be obtained.

Agent

$$Object$$
$$goals \neq \{\}$$

AutonomousAgents

$$Agent$$
$$motivations \neq \{\}$$

Thus, agents are objects with a not-empty set of goals, and autonomous agents are motivated agents.

2.2 Macondo platform

Macondo[1] [Ciancarini *et al.*, 2000], is an open source platform for distributed Java applications based on mobile agents and coordination. The principal attributes of Macondo are mobility and coordination. The mobile agent model is based on the concepts of *Agent* and *Place*. Agents are autonomous processes that move through different places for their own purposes. The place is the environment that hosts agents.

Macondo, a flexible, simple and open source, is a valid ground structure which can be developed to implement new agent systems with different specifications and functionality.

Coordination between agents is achieved with MJada, an extension of Jada [Ciancarini and Rossi, 1996]. Jada is a Linda-like [Gelernter, 1985] coordination language based on a set of classes used to access an object space. The object space is an object container with a set of methods for accessing its contents. MJada uses a shared space to exchange tuples and improves Jada by giving support to mobile agents.

[1] Macondo and Mjada were developed at the University of Bologna. Macondo is being expanded at the University of Camerino.

3 The reactivity for an agent

Often, an agent architecture consists of three layers: 1) the Reactive Layer by which an agent can respond in a timely and efficient manner to environment evolution; 2) the Planning Layer, used to support an agent's long-term planning activity, and 3) the Social Layer which supports multi-agent social reasoning and group planning. The layered architecture can be refined or partially modified, but the three main targets remain the same: Reaction, Planning and Social Reasoning. In the literature, several architectures adopt the three-layer approach. InteRRaP [Müller, 1996] and TouringMachine [Ferguson, 1992] are two examples. Other three-layer approaches may be found in [Gat, 1998; Bonasso *et al.*, 1996].

Usually, the reactive component generates actions that could potentially conflict with actions generated by other components; for example, the planning layer may decide to turn left to reach home, while the reactive layer may decide to go right to avoid an oncoming car. The agent must choose which of the two actions to perform, and in this case the conflict-resolution process can be critical. The agent must immediately decide upon the best action or his car may be hit. Therefore, the conflict resolution component must be very efficient. Actions coming from reactive layers must be generated and executed very quickly. Thus, a good model will use condition-action rules. Each rule has two parts: the first containing a condition which the environment can satisfy, the second containing a set of actions which an agent performs as a reaction to the environment evolution. If the condition is true, the action is executed. Conflict may also arise if the environment simultaneously satisfies more than one condition. In this case each rule will generate the associated set of actions and the conflict-resolution component must again choose the right behaviour.

4 The agent reactive component

4.1 Basic schemas

An event is a significant occurrence in the world.

$$Event : \mathbb{P} Attribute$$

The event is defined by a set of *Attributes* which describe the occurrence. The event must be generated only if the environment has those specific attributes. Every environment constitutes a set of possible events that can be generated, with the exception of the empty set. An environment with an empty set of events is a model for a platform that cannot generate events.

Env

$$environment : Environment$$
$$entities : \mathbb{P} Entity$$
$$envpossevents : \mathbb{P} Event$$

$$environment \neq \{\}$$
$$\bigcup \{e : entities \bullet e.attributes\} \subseteq environment$$

The Env schema provides a description of the world. Therefore, the world is an *environment* populated with *entities*. Each world has a set of events that can be generated.

A generating-events environment is not sufficient to manage reactivity. Reactions must also be defined.

A reaction is an appropriate response to a specific event. Thus, before a reaction can be described, an event and a set of actions associated with it are required.

Reaction

$$event : Event$$
$$re\_action : \mathbb{P} Action$$

$$re\_action \neq \{\}$$

4.2 Reactive agent schema

The reactive component of an agent, or a single reactive agent, can be considered a refined object. The goals and motivations that characterise agents and autonomous agents, respectively, are irrelevant. The only requisite is the potential for action. Objects have this potential. So, by this definition, reactive agents are objects.

ObjectReaction

$$Object$$
$$objectpossevents : \mathbb{P} Event$$
$$objectinterestingevents : \mathbb{P} Event$$
$$objectactualevents : \mathbb{P} Event$$
$$objectreactions : \mathbb{P} Reaction$$
$$objectreactionsgen : \mathbb{P} Event \rightarrow \mathbb{P} Action$$

$$objectinterestingevents \subseteq objectpossevents$$
$$objectactualevents \subseteq objectinterestingevents$$
$$\forall r, r' : Reaction \mid (r \in objectreactions) \wedge (r' \in objectreactions) \wedge (r.event = r'.event) \bullet r.re\_action = r'.re\_action$$
$$\bigcup \{r : Reaction \mid r \in objectreactions \bullet r.event\} = objectpossevents$$
$$\bigcup \{r : Reaction \mid r \in objectreactions \bullet r.re\_action\} \subseteq capabilities$$
$$\forall e : Event \mid e \in objectpossevents \bullet (\forall r : Reaction \mid (r \in objectreactions) \wedge (r.event = e) \bullet objectreactionsgen\ \{e\} = r.re\_action)$$

objectpossevents is the set of all events that can be intercepted. Other events are meaningless for the reactive component. An event that is not an element of *objectpossevents* cannot be understood. *objectinterestingevents* is used to denote the subset of all possible events pertinent to the present situation. This subset, obviously, is modified in relation to changes that occur in the environment. For example we might wish to be warned of an incoming obstacle while we are in motion, but are no longer interested in this event when we are at rest.

The reactive agent must know how to react properly to each Event. Therefore, the *objectreactions*

set provides a reaction for each possible event. Furthermore, the reactive agent must be able to perform all actions required in response to an event. This is why all *re_action* are included in the *capabilities*.

objectactualevents contains events generated within the environment. Each event included within the *objectactualevents* is awaiting management.

Finally, *objectreactionsgen* selects the corresponding reaction for an event from the *objectreactions* set. The *objectactualevents* is dynamically updated by removing all managed events, as illustrated in the following.

ΔObjectReaction

$$
\begin{array}{l}
ObjectReaction \\
ObjectReaction' \\
notmanagedreactions : \mathbb{P} Event \rightarrow \mathbb{P} Event \\
\hline
objectpossevents' = objectpossevents \\
objectactualevents' = \\
\quad notmanagedreactions\ objectactualevent
\end{array}
$$

The ObjectState schema is the schema for an object having acting capabilities.

ObjectState

$$
\begin{array}{l}
EntityState \\
ObjectAction \\
willdo : \mathbb{P} Action \\
\hline
willdo = (objectactions\ environment) \\
willdo \subseteq capabilities
\end{array}
$$

The variable *willdo* specifies the next action the object will take. If we require a reactive agent or, even better, a reactive object, we must add the ObjectReaction schema to the ObjectState schema. This inclusion leads to the ReactiveAgent schema.

ReactiveAgent

$$
\begin{array}{l}
ObjectState \\
ObjectReaction \\
resolveconflict : \mathbb{P} Action \rightarrow \mathbb{P} Action \\
willdo : \mathbb{P} Action \\
\hline
willdo = \\
\quad resolveconflict(\ (objectactions\ environment) \cup \\
\qquad (objectreactionsgen\ objectactualevents))
\end{array}
$$

Both *objectactions* and *objectreactionsgen* occur in the action-generating process. *objectactions* is used in the planning process, while *objectreactionsgen* is used in short term activity. At times a conflict resolution process, activated using the *resolveconflict* function, may be necessary to determine which action to perform. At the object level, the planning process is quite simple, so the difference between the *objectactions* and the *objectreactionsgen* are nearly irrelevant. In *Agent* and *AutonomousAgent* schemas the action selection process is extremely complex (*goals* and *motivations* are also used in the decision making process), so short time response and long term planning are fundamentally different activities.

4.3 Environment and Agents

The Env schema does not explicitly show the presence of reactive agents in the environment. Therefore, this schema must be refined. The following ReactiveEnv schema proposes a potential refinement.

ReactiveEnv

$$
\begin{array}{l}
Env \\
reactiveagents : \mathbb{P} ReactiveAgent
\end{array}
$$

The environment during its evolution notifies different events. This activity is formalised in the ReactiveEnvNotify schema below.

ReactiveEnvNotify

$$
\begin{array}{l}
\Delta ReactiveEnv \\
notify : Environment \rightarrow \mathbb{P} Entity \rightarrow \\
\qquad Environment \rightarrow \mathbb{P} Entity \rightarrow \\
\qquad ReactiveAgent \rightarrow \mathbb{P} Event \\
\hline
envpossevents = envpossevents' \\
\forall e : ReactiveAgent | e \in reactiveagents' \bullet \\
\quad e.objectactualavents = \\
\qquad notify\ environment\ entities \\
\qquad\quad environment'\ entities' \\
\qquad\quad e \\
\forall env_1, env_2 : Environment; \\
\quad ent_1, ent_2 : \mathbb{P} Entity; e : ReactiveAgent \bullet \\
\qquad notify\ env_1\ ent_1\ env_2\ ent_2\ e \subseteq envpossevents'
\end{array}
$$

The variable *envpossevents* is an invariant because it is constant over the time. *notify* selects all events that must be communicated to a particular agent. Events recognition is based on previous and present states of environment and entities, and also on the target reactive agent status.

Changes in the variable *objectactualevents* immediately cause changes in the variable *willdo* because the following relation is always true
$willdo = (objectactions\ environment) \cup$
$(objectreactionsgen\ objectactualevents)$.

Differences in the decision making process continue to be explicit in the *ReactiveAgentState* and in the *ReactiveAutonomousAgentState* schemas.

ReactiveAgentState

$$
\begin{array}{l}
AgentPerception \\
AgentAction \\
ReactiveAgent \\
posspercepts, actualpercepts : View \\
\hline
actualpercepts \subseteq posspercepts \\
posspercepts = \\
\quad canperceive\ environment\ perceivingactions \\
actualpercepts = willperceive\ goals\ posspercepts \\
perceivingactions = \{\} \Rightarrow posspercepts = \{\} \\
willdo = resolveconflict(\\
\quad (agentactions\ goals \\
\qquad\quad actualpercepts\ environment) \cup \\
\quad (objectreactionsgen\ objectactualevents))
\end{array}
$$

ReactiveAutonomousAgentState

$$
\begin{array}{l}
ReactiveAgentState \\
AutonomousAgentPerception \\
AutonomousAgentAction \\
\hline
willdo = resolveconflict(\\
\quad (autoactions\ motivations\ goals \\
\qquad\quad actualpercepts\ environment) \cup \\
\quad (objectreactionsgen\ objectactualevents))
\end{array}
$$

agentactions uses *goals* and *actualpercepts* to generate actions. *autoactions* uses also *motivations*.

objectreactionsgen uses only *objectactualevents*. The differing complexities of these functions includes more than simply the number of subjects. The *objectreactionsgen* only take the pre-estabilished set of actions in the correspondent reaction, using it to act. *agentaction* and *autoaction* must generate new plans and new sets of actions, which are costly activities.

5 The reactive module for Macondo

In this section we discuss the implementation strategy followed to implement the reactive component for the Macondo platform.

The first version of Macondo does not generate events. The *Place* hosts agents and provides an interface for resources. In the proposed extension, the *Env* is a refinement of the *Place* with event generation capabilities. An important component of the *Env* is the *Events Monitor*, which tests whether the environment satisfies certain conditions in order to generate events. The *EventsMonitor* activates different threads to check different possible situations (Fig.1). All events to be recognized are distributed

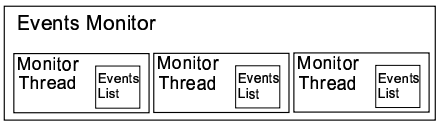

Figure 1: Events Monitor

among the *Monitor Threads* in order to minimize activity for each thread. When an event is to be generated, the *Monitor Thread* that senses it alerts the *Events Monitor*.

The reactive object component of each agent includes a list of interesting events. This list is used by the *Events Monitor* to establish how many events must be monitored and how many *Monitor Threads* are required. Clearly, it is not necessary to check the environment for events that are not in any interesting list, because no agent will react to this event.

Env uses different *Event Notifier* threads to notify events. Each thread is associated with a group of agents. The association is made in order to form homogeneous groups for events notification. For example, there will be a thread for the notification of events $e1$ and $e2$, a thread for each event $e3$ and a thread for events $e4$ and $e5$. Any agent will be present in a group, if in its interesting events list there is such event. Obviously, an agent can be present in more than one group. Therefore, when the event $e2$ is generated only the corresponding *Event Notifier* is alerted and the notification is made only to agents in its group Fig. 2.

The association of an event to only one (or to a few) thread(s) minimises work among all threads. If we wish to minimise notification time, we can distribute an event generation among all threads. In this way we achieve maximum parallelism, but also maximum resource consume, Fig. 3.

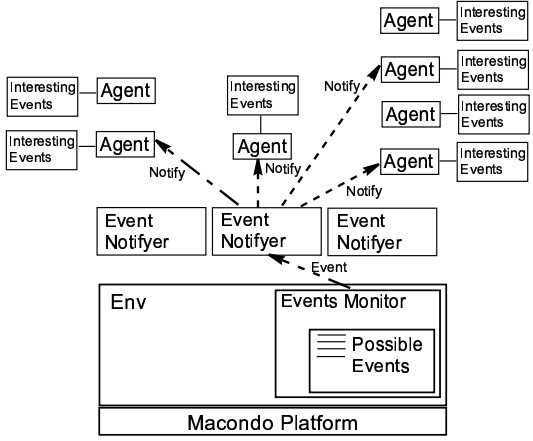

Figure 2: Event Notification

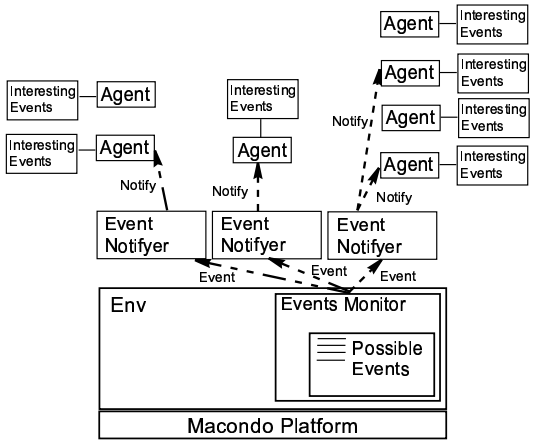

Figure 3: Parallel Event Notification

When agents are stimulated by an *Event Notifier*, they must update the actions they may have to execute. Thus, agents immediately adjust their behaviour in response to the events they receive. This is agent reaction.

An incoming agent must ask permission of the *Env* to be executed, so that the *Env* can ensure that there are not too many agents at its location in order to avoid the collapse of the system because of the presence of too much agents.

To estimate the efficiency and performance of the system, tools for testing and validation are necessary. Agents' performances must be estimated with and without the reactive component, so that it becomes clear when reactivity increases performance and when reactivity is not useful. Testing is also necessary to understand whether situations exist where the systems collapse due to the rapid growth of events generation and how these situations should be managed.

Setting up the system with these tools facilitates the assessment of the best configuration of all components in all situations and prevents critical status.

6 Conclusions

In this article, we have extended the SMART agent framework giving explicit representation to the reactive component. The description of the new included schema is not at an abstract level, but approaches an implementation level because our goal was to provide the bases for future software agent testing and validation techniques.

The formal specification proposed in this paper for the agent's reactive component takes advantage of software environments. The events generation is left to the environment; agents must only decide which events are of interest to them so that only interesting events are generated and agents are never alerted to useless occurrences. Agents are free to devote time to other activities and so improve their performance. It is better to have a single component checking for events (the environment), than thousands of agents.

If there are too many agents or strict time constraints, the environment can change its policies for events and thread generation. With activities concentrated in only a few threads, some agents could be alerted out of synch; if activities are distributed on all threads, the system could be over-stressed so that the specific context and the specific situation could lead to the definition of correct policy.

The proposed implementation strategy and the proposed formal specification have been used to extend a Macondo platform by adding a new reactive component. The resulting system allows for different configuration of all components for any possible context and for management of the reactivity of autonomous agents in a formal, robust and well known manner. In a future project, we will attempt to define a formal technique for the testing and validation of the system's reactive component.

Future work also will include a more accurate formal description of events management in order to establish precisely which is the better configuration of the system in each possible situation. Ascertaining resulting performances relating to events distribution among all threads is of utmost interest.

References

[Ahuja *et al.*, 1986] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[Bonasso *et al.*, 1996] R.P. Bonasso, D. Kortenkamp, D.P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. In *Intelligent Agents II*, pages 187–202. Springer-Verlag, 1996.

[Cabri *et al.*, 2000] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[Camurri and Coglio, 1998] A. Camurri and A. Coglio. An architecture for emotional agents. *IEEE Multimedia*, 5(4):24–33, 1998.

[Ciancarini and Rossi, 1996] P. Ciancarini and D. Rossi. Jada - coordination and communication for java agents. In *Second International Workshop on Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, 1996.

[Ciancarini *et al.*, 2000] P. Ciancarini, A. Giovannini, and D. Rossi. Mobility and Coordination for Distributed Java Applications. In S. Krakowiak and S. Shrivastava, editors, *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, pages 402–425. Springer-Verlag, 2000.

[d'Inverno and Luck, 2001] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer-Verlag, 2001.

[Ferguson, 1992] Innes Andrew Ferguson. Touringmachines: An architecture for dynamic, rational, mobile agents. Technical report, University of Cambridge, 1992.

[Gat, 1998] E. Gat. On three-layer architectures, 1998.

[Gelernter, 1985] D. Gelernter. Generative communication in linda. *ACM Computing Surveys*, 7(1):80–112, 1985.

[J.M.Spivey, 1992] J.M.Spivey. *The Z Notation, A Reference Manual*. Prentice-Hall, 2 edition, 1992.

[Larsson and Hayes, 1998] J. E. Larsson and B. Hayes. Guardian:an intelligent autonomous agent for medical monitoring and diagnosis. *IEEE Intelligent Systems*, 13(1):58–64, 1998.

[Liscano *et al.*, 1995] R. Liscano, R. E. Fayek, A. Manz, E. R. Stuck, and J. Tigli. Using a blackboard to integrate multiple activities and achieve strategic reasoning for mobile-robot navigation. *IEEE Intelligent Systems*, 10(2):24–36, 1995.

[Luck and d'Inverno, 2001] M. Luck and M. d'Inverno. A conceptual framework for agent definition and development. *The Computer Journal*, 44(1):1–20, 2001.

[Müller, 1996] J. P. Müller. *The Design of Intelligent Agents: A Layered Approach*, volume 1177 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[Norvig and Russell, 1995] Peter Norvig and Stuart Russell. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[Wooldridge and Jennings, 1994] M. Wooldridge and N. R. Jennings. Agent theories, architectures and languages: A survey. In *Intelligent Agents, ECAI-94 Workshop on Agent Theories,Architectures and Languages*, pages 1–39. Springer-Verlag, 1994.