LOCUS: An environment description language for JASON

**Ramon Fraga Pereira, Maurício Cecílio Magnaguagno,
Felipe Meneguzzi, Anibal Sólon Heinsfeld**

[1]School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

{ramon.pereira, mauricio.magnaguagno, anibal.heinsfeld}@acad.pucrs.br

felipe.meneguzzi@pucrs.br

Abstract. JASON *is an* AGENTSPEAK *interpreter for multi-agent system development, in which agents are described in the* AGENTSPEAK *language. Therefore, we only have to describe the agent behavior, but the environment does not follow this style, it requires a Java description of how the actions and perceptions operate. This choice of implementation guarantees that even complex environments can be created for* JASON*, but it requires knowledge about both Java and* JASON*'s Application Programming Interface (API). In this paper we aim to fill the gap between the languages with an* AGENTSPEAK*-like description of the environment. To overcome this gap we propose* LOCUS*, a source-to-source compiler which generates a Java description of the environment for* JASON*, providing the user with an easier starting point to create complex environments with a consistent description for both the agents and the environment. The output of* LOCUS *can be further modified if required, not limiting the user to the features already provided by* LOCUS.

1. Introduction

A multi-agent system consists of a population of autonomous computational entities situated in a shared structured environment. From this definition, we emphasize the importance of the environment, agents are not isolated entities since they share the same space. In several situations we want to simulate agents that perceive, reason, and act to achieve their goals within an environment. To describe agent behavior concisely a specific paradigm is used, Agent Oriented Programming (AOP) [Shoham 1993]. This behavior must match the restrictions of the environment to enable an agent to act successfully while pursuing its interests. Therefore, it is extremely important to model the environment correctly in multi-agent systems. However, most work on AOP, focuses on the formal description of the agents, rather than the environment. Some frameworks opt to use an AOP language to better describe the agents, while the environment remains described in an imperative language. Many researchers neglected to integrate the environment as a primary abstraction in models and tools for multi-agent systems, and much research is concerned exclusively with agents.

In this paper, we describe a declarative approach to create virtual environments in an AGENTSPEAK-like language. We target JASON, a platform for the development of multi-agent systems, due to the familiarity with the platform and current use of AGENTSPEAK to describe agent behavior. Currently, the environment specification in JASON must

be encoded in Java, which mix levels of abstraction used between agent and environment description. In this way, we propose LOCUS[1], a source-to-source compiler that converts an AGENTSPEAK-like description of the environment into a Java version with the corresponding bindings to the JASON API[2]. Although there is no gain in expressivity of the environment behavior, our aim is to allow designers to write simpler code that is easier to read. Without the common worries of API usage, the user is able to focus on what matters the most, i.e. the correct operation of the desired environment. Thus, our main contribution is an approach to describe the environment without mixing abstractions while maintaining the possibility to do so if required. We demonstrate the applicability of our approach through a subset of the standard JASON application examples. Our proposed environment description is concise and yields the Java description of the environment based on the JASON API, when the description is correct, and warns the developer otherwise.

This paper is organized as follows. Section 2 reviews the background on agents and environments, AGENTSPEAK, and JASON. Section 3 presents LOCUS and some practical examples of its usage. In Section 4 we address related work on environment description for multi-agent systems. In Section 5 we conclude with final considerations and give directions for future work.

2. Background

This section introduces essential background on agents, and important concepts related to multi-agent system environments.

2.1. Agents and Environments

Agents act through actuators and perceive through sensors within an environment [Russell and Norvig 2009]. Actuators and sensors allow agents to interact with the environment, these interactions are represented by actions and perceptions of the agent [Weiss 2013]. More specifically, actions are executed through actuators and produce an output that affect the environment. Actions are fundamental because they represent the mechanism through which agents change the environment. Perceptions are given as input to agents through sensors from the environment. In general, the interactions with the environment describe the agent behavior, for example, given an input of perceptions the agent can reason to act within an environment in order to achieve a desired goal [Wooldridge 2009].

Multi-agent environments are typically modeled depending on a variety of properties, for example: the environment can be deterministic or non-deterministic; fully or partially observable; and interactions can happen instantly or with a time duration. These properties tend to impact on how these agents interact with the environment, and consequently on how agents have to reason in order to act in an environment. Agents commonly represent the environment internally as a belief base, i.e, what the agents believe to be true from the interactions with the environment and other agents. Conversely, the environment contains an internal representation of the result of interactions between agents as a state, holding the properties of every object.

[1]The word locus is latin for place.
[2]http://jason.sourceforge.net/wp/

2.2. AGENTSPEAK(L) and JASON

AGENTSPEAK(L) is a programming language based on logic programming for the Belief-Desire-Intention (BDI) agent architecture which provides support for events and actions [Rao 1996]. The BDI architecture is used to provide components that bring about a notion of "mental state" for agents. It is characterized by implementation of computational analogues of human beliefs, desires, and intentions. The current state of the agent, which is a model of itself, its environment, and other agents, can be viewed as its current belief state. The states which the agent wants to accomplish are desires, based on its external or internal motivations. The adoption of programs to satisfy such stimuli can be viewed as intentions. The behavior of the agent, i.e, its interaction with the environment is encoded by the programs written in AGENTSPEAK(L). Thus, an AGENTSPEAK agent is created by the specification of a set of base beliefs and a set of plans[3] [Bordini and Hübner 2005].

JASON is an AGENTSPEAK interpreter for multi-agent system development, in which agents are described in an extension of the AGENTSPEAK language, but the environment is described in Java [Bordini et al. 2007]. JASON infrastructure is developed in Java and allows the customisation of the agent behaviors and reasoning. It provides three main constructs for agent programming and reasoning: beliefs, goals and plans, each described in turn below.

2.2.1. Beliefs

In JASON, beliefs are represented as predicates and stored in a collection called belief base. Unlike classical logic, beliefs are not an absolute truth. Each agent has an individual belief base and each stored belief is only true for its agent. As the logic programming language Prolog, a belief is represented by an atom (a sequence of characters starting with a lower-case letter) or a structure. Structures are useful to represent complex beliefs composed by an atom followed by arguments, for example door(closed).

JASON handles negation in two ways: using 'not' and $\sim$ operators. The first will lead the interpreter to result *true* about some formula if it cannot be derived using beliefs and rules of the agent. The second is called *strong negation* and brings the notion that a belief is explicitly false.

A difference from PROLOG syntax is the belief annotations. Annotations provide meta-level information about beliefs, like the source of beliefs. The source is annotated automatically in each belief by JASON, but custom annotations can be created in order to provide useful information for agent reasoning, such as the moment that the agent adds a belief. However, annotation are meaningless to the JASON interpreter, so the programmer must develop an expected behaviour for the annotations. Actually, annotations could be represented as other beliefs, but JASON's syntax provides a better readability and linkage of information.

Another structure provided in JASON is the Rule. A rule is a logic formula that the agent uses its belief base to reason about its truth. Rules bring brevity to the code, since it is not necessary to always represent the formula that is entailed by the rule.

[3]Plans represent a sequence of actions that an agent is able to perform in a environment to achieve a goal.

```
1 triggeringEvent : context <- body.
```

Listing 1. The three plan parts in JASON.

2.2.2. Goals

In JASON, goals represent the properties of the states of an agent's environment that the agent wishes to bring about. JASON has two types of goals: achievement goals and test goals. Achievement goals ares denoted by an exclamation mark and a predicate. If the goal is declarative, the agent believes that the predicate is not true and will act in order to modify the state of its environment to make it true, otherwise it is a procedural goal, the proposal of the goal is basically to group actions together, which is useful when these actions are often used. Test goals are normally used to retrieve information from belief base and are denoted by interrogation mark. When an agent adopts a goal, it executes a Plan, that is composed by a sequence of actions to change environment and agent states.

2.2.3. Plans

A plan in JASON consists of three parts: the triggering event, the context and the body, Listing 1. The triggering event can be an addition or a deletion of beliefs or goals, due to environment changes. So, the plan will be executed as a sequence of actions as a consequence of such event. The context is used to check the current circumstance in order to determine if the plan is expected to succeed. Due to different circumstances, a plan that better fits in the current state is chosen according to its context specified by agent designers. The body is composed by a sequence of actions, containing sub-goals that the agent must assume to proper handle the triggering event.

3. LOCUS Environment Description Language

LOCUS implements the counterpart of Agent Oriented Programming (AOP), targeting a concise and correct environment description. In the same way an agent requires a belief base to create an internal representation of the world, the environment requires a structure to hold its own current state. The difference is the point of view, the agent beliefs have a known or unknown degree of truth, due to the sensing capabilities of the agent who perceives part of the world at specific moments. The environment represents the entire state without errors, it contains the ground truth, and communicates parts of this truth as perceptions to the corresponding agents. The agent with the perceptions and reasoning may act, calling an environment action. The environment deals with the outcome of actions, those actions can only happen if the preconditions are satisfied, i.e, the agent class or the agent accuracy. With those concepts in mind, we create AGENTSPEAK-like constructs to represent them.

In order to be easier to describe we divided the environment in three main parts: an initial state (*init*), actions (*beforeActions, +action, afterActions*), and a stop call (*stop*), they can be seen in Listing 2. Although the environment parts can be defined in any order, we believe this order to be easier to understand. Firstly, we define an initial setup, in which the initial state of the environment is defined along with the perceptions they create.

```
1  init                                    <- body.

2  beforeActions                           <- body.
3  +action(name[, terms]) : context        <- body.
4  afterActions                            <- body.

5  stop                                    <- body.
```

Listing 2. The environment main parts.

```
1  +percept(agent|all, predicate[, terms]) : context.
2  -percept(agent|all, predicate[, terms]) : context.

3  +state(predicate[, terms]);
4  -state(predicate[, terms]);
5  -+state(predicate[, terms]);
```

Listing 3. Modifying perceptions and the environment state.

This creates a common starting point to the simulation. Secondly, we define how the actions affect the environment and the restrictions imposed on specific action executions. Additionally, in the second part, it is possible to define what happens before and after the execution of actions. This part is essential to keep environment and agents interacting, modifying the state and which perceptions happen during the simulation. Finally, the stop setup defines what happens at the end of the simulation. Although not explored so far we believe to be important to log data at the end of simulation.

The actions are the main part, as they may be called several times per execution. The *beforeActions* controls what happens before any action takes place, usually clearing a set of percepts for some or all agents. After that the action that was called by name by an agent takes place and any term given as an argument is forwarded. If the context is satisfied the body of the action is applied. We provide macros to test agent name and agent class, **agentName** and **agentClass** respectively, to make it easier to test agent attributes.

Inside each part we find their respective body, a set of commands to describe what happens with the environment. The commands in Listing 3 can be used in the body of the following statements: *init, actions, before and after actions, and stop*. We focus on commands about perceptions and state at the current stage of development. It is possible that perceptions are broadcast to all agents or are restricted to a specific agent or subset of agents. However, the context must be true to apply the desired perception, like a sensor working properly and in range to capture the data. In order to handle the state of the environment, it is possible to specify features of the environment as propositions to be added or deleted from the state. The only impact of this approach is the generic structure used to hold the state configuration, while handmade versions of the environment would use tailored structures, like a single boolean variable to store a door state. The symbol plus ($+$) represent addition, and the symbol minus ($-$) represent deletion of a feature. These symbols are used as a prefix of the statement **percept**. Moreover, to update an environment state, the symbol ($-+$) is declared as a prefix of the statement **state**. In Listing 3 we exemplify how to add, remove, and update perceptions and states in LOCUS.

```
1 +!locked(door)[source(paranoid)] : ∼locked(door) <- lock.
2 +!∼locked(door)[source(claustrophobe)] : locked(door) <- unlock.
```

Listing 4. Porter agent.

```
1 +locked(door) <-
2     .send(porter,achieve,∼locked(door)).
```

Listing 5. Claustrophobe agent.

```
1 +∼locked(door) <-
2     .send(porter,achieve,locked(door)).
```

Listing 6. Paranoid agent.

We also use the plus symbol to add actions to the environment, further investigation is required to see how useful it would be to remove actions using a $-action$ versus using a context.

3.1. Examples

In this section we use examples in JASON to show how our source-to-source compiler can be used. Those examples describe small environments, but handle with state control and agent perception. We make use of the room application, one of the JASON examples, and Bakery application, an example created specifically to show how our approach handle with different environments.

3.1.1. Room

This application consists of a room with a door, and contains the following agents: a porter, a claustrophobic and a paranoid. The only object in this room is the door that may be closed or not. The porter (Listing 4) is the only agent that modifies the door state, using lock and unlock actions, while all agents can perceive the current state of the door. When the claustrophobic agent (Listing 5) perceives the door closed the only option is to ask the porter to open the door, after all this agent does not like to be confined. The paranoid (Listing 6) does not share the same fear, instead this agent prefers to be safely closed inside the room. Once the paranoid perceives the door opened the only option is to ask the porter to close it. The porter only does as told.

The environment is extremely simple: it contains two different perceptions and two different internal actions about the same door. In the Prometheus diagram [Winikoff and Padgham 2004], in Figure 1, we see only one agent able to interact with the door, the porter. Yet, in the current version of JASON these actions have no test towards the agent class or name. This lack of test makes any agent eligible to close the door, which creates no problem for this small example, but shows how hard it can be to follow a specification when you need to code in two different styles. Our description in LOCUS (Listing 7) covers almost the entire Prometheus design, it only lacks a description of the message passing mechanism between agents, since message passing is done in JA-

```
1  init <-
2      +state(doorLocked);
3      +percept(all, locked, door).

4  beforeActions <-
5      -percept(all).

6  +action(lock) : agentClass(porter) <-
7      -+state(doorLocked).

8  +action(unlock) : agentClass(porter) <-
9      -+state(~doorLocked).

10 afterActions <-
11     +percept(all,  locked, door) : state( doorLocked);
12     +percept(all, ~locked, door) : state(~doorLocked).
```

Listing 7. LOCUS description for Room application

SON without environment restrictions. The generated Java code matches the original Java environment in JASON – it is able to replace that code without further modifications.

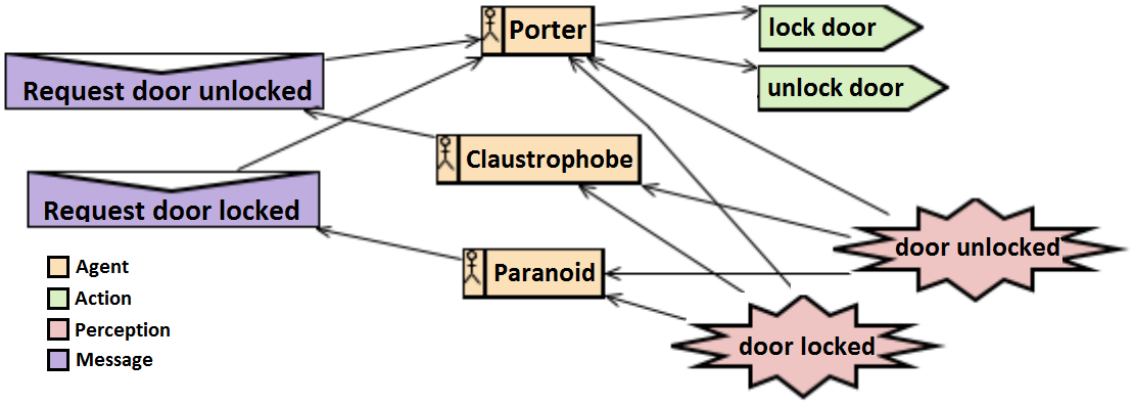

Figure 1. An overview of the room example.

3.1.2. Bakery

In our second example we use a bakery application, containing two types of agents: a boss and one or more bakers. The Prometheus diagram of this application is shown in Figure 2. Our goal with this example is to create and perceive the existence of items in the environment. The boss (Listing 8) perceives when there is one item missing from the shelf, and since the boss likes to maintain the shelf full, pins a note with the task to bake more of some item. The baker (Listing 9) perceives there is a new task, and starts baking the new item. When finished the task, the new item is put directly in the shelf. The boss can now perceive the task was done. We could continue and add an agent to sell and other to buy items randomly, and keep the system in a loop, but we believe this subset of the application already shows how useful our source-to-source compiler can be. Listing 10 contains the environment, with several perceptions targeting the boss

```
+~have(C) <- pinTask(C).
+have(C) <- .print("Done␣", C).
```

Listing 8. Boss agent.

```
+newTask(X) <- bake(X).
```

Listing 9. Baker agent.

```
init <-
    +state(~have(pie));
    +state(~have(cake));
    +state(~have(donut));
    +percept(boss, ~have, pie);
    +percept(boss, ~have, cake);
    +percept(boss, ~have, donut).

beforeActions <-
    -percept(all).

+action(pinTask, C) : agentName(boss) <-
    +percept(all, newTask, C).

+action(bake, C) : agentClass(baker) <-
    -+state(have, C).

afterActions <-
    +percept(boss, ~have, pie)   : state(~have(pie));
    +percept(boss, ~have, cake)  : state(~have(cake));
    +percept(boss, ~have, donut) : state(~have(donut));
    +percept(boss,  have, pie)   : state( have(pie));
    +percept(boss,  have, cake)  : state( have(cake));
    +percept(boss,  have, donut) : state( have(donut)).
```

Listing 10. LOCUS description for Bakery application

and explicitly declaring the existence or lack of each item. It is possible to observe the redundancy that appears in the **afterActions** block, in which we are testing both positive and negative cases. The lack of an ”else” construct is not a problem, but takes repeated conditions to take care of. The good side effect is to have perceptions with independent causes, but we hope to address the problem of code repetition as bigger applications reveals them.

4. Related Work

Regarding the environment description for multi-agent systems, we highlight two related work: ELMS (Environment Description Language for Multi-Agent Simulation) and CArtAgO (Common Artifact infrastructure for Agent Open environments).

CArtAgO provides the infrastructure to develop environments based on the A&A (Agent and Artifacts) meta-model [Ricci et al. 2010, Ricci et al. 2011]. This meta-model describes three basic abstractions. First, the agents that represent pro-active components

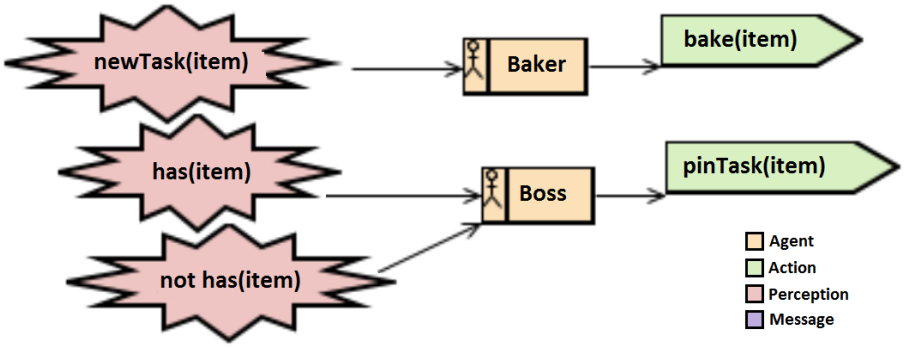

Figure 2. An overview of the bakery example.

of the system. Second, the artifacts, which represent passive components of the system and support agents' activities. Third, the workspaces, that provide a notion of topology for the environment. In CArtAgO, artifacts are not autonomous and can not follow their own course of action, serving as tools for agents. The programming of CArtAgO artifacts is made using a Java-based API. This API provides Java annotations and classes to make accessible specific methods to JASON, aiming at the interaction of agents with the artifacts. CArtAgO is part of the framework JaCaMo. JaCaMo [Boissier et al. 2013] combines the environment description of CArtAgO with JASON and Moise, an organisational model for multi-agent systems. This union of technologies provides a robust development environment, allowing cooperative and distributed agent behaviours. However, CArtAgO and Moise require the use of different languages and paradigms, which can difficult the development and the linkage of design concepts.

In order to describe the environment, [Okuyama et al. 2004] propose ELMS to use a markup language syntax, namely XML. Due to generality of XML, the authors defined a set of elements and attributes to describe the parts covered by the work. First, the structures to define agent bodies specify "physical" aspects of the agents, such as the properties that may be perceived by other agents and the actions that agents can perform in the environment. Second, the work provides a structure to define 2D or 3D grids, if the designer has chosen to have one. Attributes can be defined for each cell of the grid to be perceived by agents. Reaction of actions also can be defined, as a response to the actions of agents and changes to the attributes. Objects can be placed on the environment via the notion of resources. Resources, as grid cells, have their own attributes and reactions. Third, some elements are defined in order to specify the initialisation process and parametrizations of the environment. As noted by the authors, describing the environment with XML can be cumbersome and may require a graphical interface to ease the development process.

5. Conclusions

As JASON's API gets more complex the direct use of Java constructs requires a good understanding of the inner workings of Java, which takes time and is often not the goal of JASON users. New users of JASON are LOCUS target audience, being able to use simpler constructs without knowledge about JASON's internal structures. Two other advantages became clear during the development process, the first is the ability to include the environment description, now much shorter, in texts without having to explain Java code first or select slices of the original description. The second advantage is to be independent

from the internal structures used by JASON and its extensions, which may prevent updates in JASON's API from breaking legacy code (as is often the case even with minor JASON updates). There is a long way before LOCUS supersedes the direct Java descriptions used today. Some applications may require specific constructs available in Java (File input, optimized structures, random outcomes from the same action) and the limitations of our solution to recreate the behavior in the current state. Those constructs could be solved right away, but then our implementation would replicate the imperative paradigm of Java, telling how to solve the problem, instead of representing the environment.

We now focus on several constructs to be added, being implemented to match JASON's AGENTSPEAK behavior, adding unification and lists to describe more complex environments than what is currently possible. The goal is to use Java as a last resort, keeping the description free of an internal architecture. In the current version we see the current models and views as a problem to our goal. Models are tailored structures to hold the state, while views are graphical interfaces to show the state. Model and view usually appear together and we search for an abstraction to give as good results as them. The model provides speed with the correct structures while the interface provides a visualization method specifically for the application. Their only problem is the complexity to maintain both working correctly, since they vary for each application. We hope to address the view in the future and create some optimizations to our generic state structure to yield a tailored one. Today the environment does not consult the agent information, like beliefs and intentions, we believe this information must remain encapsulated. Maybe a specific application will require an extension to do so. Another discussion that we had is related to time-based events, happening at some point in time or with a duration, but we have not explored this enough to create an abstraction. The current set of commands is small, but shows it is possible to describe the environment without mixed levels of abstraction while opening the subject of an AGENTSPEAK-like language to describe environments. The LOCUS project[4] is available for download in the GitHub website. In this website we provide instructions on how to use the LOCUS description language to create environments for JASON.

References

[Boissier et al. 2013] Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent Oriented Programming with JaCaMo. *Sci. Comput. Program.*, 78(6):747–761.

[Bordini et al. 2007] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[Bordini and Hübner 2005] Bordini, R. H. and Hübner, J. F. (2005). BDI Agent Programming in AgentSpeak using Jason. In *Proceeding of 6th International Workshop on Computational Logic in Multi-agent Systems (CLIMA VI). Volume 3900 of LNCS*, pages 143–164. Springer.

[Okuyama et al. 2004] Okuyama, F. Y., Bordini, R. H., and da Rocha Costa, A. C. (2004). ELMS: An Environment Description Language for Multi-agent Simulation. In *Envi-*

[4] https://github.com/Maumagnaguagno/Locus

ronments for Multi-Agent Systems, First International Workshop, E4MAS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers, pages 91–108.

[Rao 1996] Rao, A. S. (1996). AgentSpeak(L): BDI Agents Speak out in a Logical Computable Language. In *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-agent World : Agents Breaking Away: Agents Breaking Away*, MAAMAW '96, pages 42–55, Secaucus, NJ, USA. Springer-Verlag New York, Inc.

[Ricci et al. 2011] Ricci, A., Piunti, M., and Viroli, M. (2011). Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192.

[Ricci et al. 2010] Ricci, A., Viroli, M., and Piunti, M. (2010). Formalising the Environment in MAS Programming: A Formal Model for Artifact-based Environments. In *Proceedings of the 7th International Conference on Programming Multi-agent Systems*, ProMAS'09, pages 133–150, Berlin, Heidelberg. Springer-Verlag.

[Russell and Norvig 2009] Russell, S. and Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

[Shoham 1993] Shoham, Y. (1993). Agent-oriented Programming. *Artif. Intell.*, 60(1):51–92.

[Weiss 2013] Weiss, G., editor (2013). *Multiagent Systems*. MIT Press, Cambridge, MA.

[Winikoff and Padgham 2004] Winikoff, M. and Padgham, L. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. Halsted Press, New York, NY, USA.

[Wooldridge 2009] Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition.