Specifying Agent Observable Behaviour*

Mirko Viroli
DEIS, Università degli Studi di Bologna
via Rasi e Spinelli 176
47023 Cesena (FC), Italy
mviroli@deis.unibo.it

Andrea Omicini
DEIS, Università degli Studi di Bologna
via Rasi e Spinelli 176
47023 Cesena (FC), Italy
aomicini@deis.unibo.it

ABSTRACT

This paper investigates the application of a formal framework for the *observation* issue in agent-based systems to the specification of the individual agent interactive behaviour. An abstract architecture is defined for agents that is based on the idea of viewing them as *observable sources* – of knowledge, services, capabilities. In this model, only the agent portion that is directly involved in managing the interactions with the environment is explicitly represented, abstracting away from agent internal and hidden details while focussing on their observable effect.

The applicability of the formal framework as a specification tool is put to test showing how different interactive behaviours can be modelled – including reactive and proactive message sending, and notification capabilities.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications; F.1.1 [**Computation by Abstract Devices**]: Models of Computation; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multi-agent systems*

General Terms

Theory, Design

Keywords

Agent architectures, Formalisms and logics, Agent-based software engineering

1. INTRODUCTION

The complexity of interaction within applications in the Internet era calls for new approaches and paradigms for reasoning about computer systems. A multiplicity of heterogeneous components interacting within phisically distributed environments make system modelling and engineering a non-trivial task, asking for new metaphors and tools.

Observation provides for a new and powerful key for modelling complex interaction patterns. As firstly discussed in [13], most interactions occurring within complex systems can be interpreted as observation-related actions: roughly speaking, *coordinators* condition observable *sources* so as to affect their behaviour and to provide *observers* of the source perception they need. This basic observation ontology can be used to model and compare seemingly different interaction patterns in a wide range of computer systems – ranging from the object-oriented to the database fields, from the coordination to the distributed computing fields.

A uniform formal framework can be built upon this simple ontology, which can be fruitfully applied to the modelling of multiagent systems, too [14]. Intuitively, agents are particularly suitable for an interpretation in terms of observation: as observers – when emphasising their situatedness and reactivity –, as coordinators – when accounting for their proactiveness and ability to act upon the environment –, as sources – when stressing their social attitude to share their capabilities and competence. Even though the interpretations of an agent as an observer or a coordinator are apparently the most natural ones, in this paper we focus on the idea of modelling agents as observable sources, which already proved to be particularly appealing [14].

In this paper, we firstly define an abstract architecture for agents, grounded on the observation pattern developed in [13], and aimed at the specification of agents observable behaviour. According to such an abstract architecture, agents are given a partial representation – as "grey boxes", so to say –, accounting for their observable portion only while abstracting away from inner details. This is likely to abate the complexity of agent representation, by focussing on the portion of an agent that is directly affected by the interaction with the environment and that determines its observable behaviour. Also, agents that cannot be fully knowable – as in the case of legacy components, or open systems – are amenable of an abstract representation, making it possible to reasoning about their cooperation with other components.

Then, a formal framework is introduced, built upon the abstract architecture and inspired by the work in [14]. In order to demonstrate the expressive power of both our architecture and formal framework, we select some of the most frequent interaction patterns occurring in agent systems, and show how they can be specified within our framework.

The remainder of this paper is organised as follows. Section 2 introduces the observation approach for agent sys-

*This work has been partially supported by MIUR, and by Nokia Research Center, Burlington, MA, USA.

AAMAS'02 July 15-19, 2002, Bologna, Italy

tems, motivating the agent abstract architecture. Section 3 presents the formal framework along with a simple application example. Section 4 puts to test the framework by encoding different kinds of interaction schemata for the individual agent. Section 5 discusses related works and Section 6 provides concluding remarks.

2. OBSERVATION AND AGENT SYSTEMS

Autonomous agents are complex software abstractions whose behaviour cannot be fully modelled and represented, in particular as far as both deliberative and social aspects are concerned. While an agent social behaviour is typically manifested through its interactions with the environment, deliberation activities are generally represented in terms of the concept of agent *mental state*, so that taking into account both aspects is a generally challenging task for software engineers. The observable source model for agents exploited in this paper is meant to provide for the proper abstractions and paradigms for tackling this problem, in that agent inner details are represented only as far as they affect the portion of the agent that is responsible for the interaction with the environment.

2.1 On the grey-box modelling approach

Sometimes, it may be interesting to model the internal behaviour of a software component focussing on its architecture and completely defining its dynamics. In this case, a *white-box* modelling approach is typically exploited that completely represents the component. Typically, this is done by providing an operational semantics that defines the relationship between the component internal dynamics and the interactions with the environment. However, in a wide set of domains – agent-based systems being a relevant case – this approach is unlikely to be effective, both because the software component could be intrinsically too complex to be fully characterised, and also because – independently from its complexity – its actual behaviour could be not completely known, as in the case of legacy systems.

Typically, this inadequacy is faced by exploiting a *black-box* approach, where the component behaviour is characterised only by means of its admissible interaction histories – that is, by the sequences of input and output communication acts the component exhibits while interacting with its environment. The most notable example of this approach is the one underlying the *observational equivalence* issue as addressed in the process algebra field [2], which provides a foundational concept to concurrent languages such as CCS [9] and π-calculus [10]. By this technique observability of interaction histories is taken as a key paradigm for stating equivalence of processes, and for assessing conformance of a specification with respect to an implementation. This approach generally promotes the idea of abstracting away from any detail concerning the actual inner machinery of a software component, which is characterised only by its observable interactive behaviour. While this framework provides a basic and valuable foundation for reasoning about interactive systems in general, and for stating properties of interest about communication protocols, it is not completely satisfactory for modelling software components of all sorts. As for the white-box approach, agents are a case where the black-box approach, too, seems not to fully succeed.

Agents are intrinsically stateful software entities. The ways they interact with the environment are not fixed once and for all, but tightly depend on their evolution over time, as well as on the decisions they autonomously take, reflecting internal deliberation, planning, and inference activities – namely, they depend on their *proactive* behaviour. Even more, very often the agent communication acts cannot be directly expressed in terms of events occurring within the agent, for autonomy making agent internal behaviour and interactive behaviour loosely coupled. For instance, an agent receiving a request for executing a given action may not guarantee that action to be actually executed, since the agent can simply ignore the request.

So, in between the black-box and the white-box approaches, a *grey-box* approach for agents can be defined, based on the idea of partitioning agent internal aspects in a part completely modelled (the *white* part) – responsible for the interactions with the environment – and a part abstracted away (the *black* part) – representing internal deliberation process, and modelled only as far as it affects the white part. In the context of grey-box modelling approaches the notion of observation is no more related to the idea of a virtual, external observer gathering information about an agent interactions with the environment [9] as in black-box approaches. Instead, observation is seen as a modelling paradigm for focussing on the agent part responsible for a behaviour that is observable by the entities of the environment. So, in this paper the term *agent observable behaviour* denotes the behaviour of the agent part determining interactions with the environment – namely, the dynamics of the agent white part and the corresponding interactive behaviour.

In order to apply the grey-box approach, however, an ontology is generally necessary so as to define how to split an agent model into a black and white part, how these parts influence each other, and how the latter provides for interaction with the environment. As argued in [16], choosing such an ontology means to define a sort of abstract architecture for agents, describing the agent behaviour in a loosely coupled way with respect to the actual agent implementation. In order to provide this architecture in the more general way, we rely on the ontology for the *observation issue* in computer systems developed in [13]. Such an ontology is focussed on representing how software components provide facilities for allowing their status and its dynamic to be perceived and possibly affected by external entities. By this technique, agents are interpreted as *observable sources*, representing the ability of sharing their knowledge and capabilities – in short, their viewpoint over the world – in terms of allowing their observable status to be perceived and altered by the environment.

2.2 Abstract Architecture

According to the general grey-box modelling approach, an agent abstract architecture is conceptually divided in two parts, the *agent (observable) core* – white part – and the *agent internal machinery* – black part –, as show in Figure 1. The agent core represents the part of the agent directly involved in how the agent behaviour can be perceived by and can affect its environment, and is modelled in terms of *position* (P) and *configuration* (C). The agent position models the part of the agent status affecting its observable behaviour; the agent configuration is the part keeping track of the interactions the agent is handling.

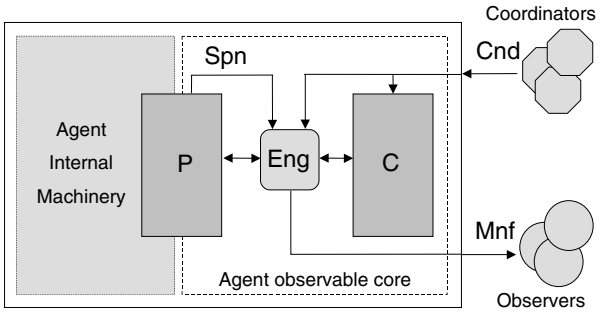

Figure 1: Agent Abstract Architecture

According to this structure, the agent dynamics can be described as follows. The agent is said to be in *equilibrium* when its core does not change. Because of the architecture shown in Figure 1, this does not model agent inactivity, but rather the internal machinery activity not being perceivable by the environment. Equilibrium can be broken in one of the two following ways. On the one hand, an external entity called *coordinator* can change the agent configuration through a *conditioning* (Cnd), modelling the agent accepting an incoming message. On the other hand, the agent can perform a *spontaneous move* (Spn), where the internal machinery activity causes a change to the place. Spontaneous moves are used to model the agent proactively willing to alter its observable behaviour. As a result of one of these two events, the agent enters in *motion*, that is, it evaluates the current state of place and configuration, producing: (i) a change on their state – namely a *place update* and a *configuration update* – and (ii) a set of messages sent out to some *observers* – called the agent *manifestations*. This evaluation is performed by a logical agent component called the *core engine* (Eng). Eventually, the agent returns in equilibrium waiting for a new conditioning or spontaneous move.

This abstract architecture is directly built over the observation ontology presented in [13], which can be suitably exploited for modelling agent features such as reactivity, proactiveness, autonomy, and social ability, as described in more detail in [14].

3. FORMAL FRAMEWORK

3.1 Notation

In this paper, sets are generally denoted by strings starting with an uppercase letter, while variables ranging over their elements by the same string (or a decoration of it) written using lowercase letters – e.g. set Any is supposed to be ranged over by variables of the kind any, any', any_1, and the like. The set of partial functions associating elements of the set Dom to elements of the set $Range$ is denoted by $Dom \mapsto Range$; the content of a function can be enumerated by the syntax $\{d \mapsto r, d' \mapsto r', \dots\}$. Function updating is defined by the operator

$$f[d \mapsto r] = f \setminus \{(d \mapsto r') \in f\} \cup \{d \mapsto r\}$$

denoting a function obtained from f by updating (or adding) $f(d)$ to r.

A finite *sequence* (or *tuple*) of elements $x_1, \dots, x_n$ is denoted by the symbol $\langle x_1, \dots, x_n \rangle$ and is considered an element of the cartesian product $X_1 \times \dots \times X_n$. A *record* containing a tuple of sets $\langle X_1, .., X_n \rangle$ – each tagged by a different label l_1,...,l_n of any sort – is denoted by the syntax $\langle l_1 : X_1, \dots, l_n : X_n \rangle$, is defined as $(\{l_1\} \mapsto X_1) \cup ... \cup (\{l_n\} \mapsto X_n)$, and has elements of the kind $\langle l_1 : x_1, \dots, l_n : x_n \rangle$.

Given any set X, symbol $\bot_X$ – or $\bot$ for short – is used to denote an exception value in set $X_\bot$ defined as $X \cup \{\bot_X\}$. Generally, variable any is supposed to range over set $Any_\bot$ too.

The set of multisets over X is denoted by $\overline{X}$, and its elements by the variable $\overline{x}$ and its variations $(\overline{x}', \overline{x}'', ..)$[1]. The content of a multiset can be specified by enumerating its elements though the symbol $\{.\}_M$ – e.g. writing $\overline{x} = \{x', x', x'', x''', ...\}_M$ – or by the union of two multisets through the binary operator '|', as in $\overline{x} = \overline{x}' | \overline{x}''$. The void multiset is also denoted by $\bullet$.

This paper borrows some techniques from the field of distributed systems, where they are used to define the behaviour of interactive systems and for generally providing a foundation to aspects such as communication and concurrency. The main mathematical structure exploited here is the *(labelled) transition system* [6], which is typically used to model software components evolving through the interactions with their environment. Formally, a transition system over the set P is defined as a triplet $\langle P, \longrightarrow, Act \rangle$, where P is called the set of processes and Act the set of *actions*. Symbol $\longrightarrow$ denotes a *transition relation* of the kind $\longrightarrow \subseteq P \times Act \times P$.

While P can be seen as the set of states of a given software component, Act represents the set of actions associated to the software component, which denotes either internal computations or interactions with environment – such as receiving or sending messages. In particular, given processes p, p' and action act, the occurrence of $\langle p, act, p' \rangle$ in $\longrightarrow$ means that process p possibly performs the action act moving to p', which is written $p \xrightarrow{act} p'$. In the context of this paper, in a transition system of the kind $\langle X, \longrightarrow, Act_\bot \rangle$ the proposition $x \xrightarrow{\bot_{Act}} x$ is always supposed to hold.

The content of a transition relation $\longrightarrow$, also called the semantics of its transition system, is given in terms of a set of rules of the kind

$$\frac{condition}{x \xrightarrow{act} x'}$$

possibly containing free variables[2]. So, given a set of rules, $\longrightarrow$ is defined as the least relation satisfying all of them for any substitution of the free variables.

Given the binary relation $\alpha \subseteq X \times Y$, the abstract notation $x \lhd \rhd_\alpha y$ is used here when $\langle x, y \rangle \in \alpha$. Correspondingly, for that relations, too, we intuitively exploit definitions "by rules" analogously to the case of transition systems.

3.2 Operational Semantics

In this paper, a labelled transition system is introduced that provides an operational semantics of the agent abstract architecture described in Figure 1.

An agent observable behaviour is expresses in terms of the following sets. P is called the set of *places*, modelling the states of the agent place. On P, two transition systems

[1]In our notation, variables x and $\overline{x}$ are considered unrelated.
[2]When there are no conditions to be specified, the upside part of the rule is completely avoided.

$\mathcal{S} = \langle P, \longrightarrow_{\mathcal{S}}, Spn \rangle$ and $\mathcal{P}u = \langle P, \longrightarrow_{\mathcal{P}u}, PU \rangle$ are built that model its dynamics. $\mathcal{S}$ defines how a place p may move to p' due to a given *spontaneous move* spn, while $\mathcal{P}u$ defines how a place p may move to p' due to a given *place update* pu. These two transition systems give semantics to the notations:

$$p \xrightarrow{spn}_{\mathcal{S}} p' \qquad p \xrightarrow{pu}_{\mathcal{P}u} p'$$

C is called the set of *configurations*, modelling the states of the agent configuration. Analogously to the set of places, two transition systems $\mathcal{C} = \langle C, \longrightarrow_{\mathcal{C}}, Cnd \rangle$ and $\mathcal{C}u = \langle C, \longrightarrow_{\mathcal{C}u}, CU \rangle$ are built on C that model the configuration dynamics. $\mathcal{C}$ defines how a configuration c may move to c' due to a given *conditioning* cnd, while $\mathcal{C}u$ defines how a configuration c may move to c' due to a given *configuration update* cu, writing:

$$c \xrightarrow{cnd}_{\mathcal{C}} c' \qquad c \xrightarrow{cu}_{\mathcal{C}u} c'$$

M is the set of manifestations the agent can produce, representing the messages an agent sends to its environment. In the context of this paper manifestations are supposed to be messages of the kind $o \uparrow v$, where $o \in O$ is an identifier for the message receiver, namely the manifestation observer, while v is the message content, which can be of any sort[3].

Finally, the couple of relations $\langle \pi, \rho \rangle$ – defined by *observable behaviour rules* – explicitly define the agent observable behaviour by relating spontaneous moves and conditionings to core changes and manifestations. In particular, the *proactiveness relation*

$$\pi \subseteq \langle P, C, Spn \rangle \times \langle PU_{\perp}, CU_{\perp}, \overline{Mnf} \rangle$$

associates to a place p, a configuration c, and a spontaneous move spn triplets $\langle pu, cu, \overline{mnf} \rangle$, orderly representing (i) how to update the place, (ii) how to update the configuration and (iii) which manifestations are to be produced. In particular, $pu = \perp_{PU}$ and $cu = \perp_{CU}$ respectively represent the place and the configuration remaining unchanged. Similarly, the *reactiveness relation*

$$\rho \subseteq \langle P, C, Cnd \rangle \times \langle PU_{\perp}, CU_{\perp}, \overline{Mnf} \rangle$$

associates triplets $\langle p, c, cnd \rangle$ to triplets $\langle pu, cu, \overline{mnf} \rangle$. As a result, while ρ describes the agent reactive behaviour, determining how conditionings lead to manifestations, π describes the agent proactive behaviour by determining how spontaneous moves lead to manifestations.

In order to define how this specification can be exploited to define the agent interactive behaviour, a transition system $\mathcal{O}bs$ is introduced as follows:

$$\mathcal{O}bs = \langle P \times C, \longrightarrow_{\mathcal{O}bs}, \{cnd \triangleright \overline{mnf}, spn \diamond \overline{mnf}\} \rangle$$

$\mathcal{O}bs$ describes how an agent equilibrium state $\langle p, c \rangle$ – called *position*, and denoted by the syntax $p \lceil c \rfloor$ – evolves through two kinds of interactions with the environment. On the one hand, the agent may send manifestations $\overline{mnf}$ due to the occurrence of spontaneous move spn – namely, by the *proactive action* $spn \diamond \overline{mnf}$ –, or the agent may react to conditioning cnd sending manifestations $\overline{mnf}$ – namely, by the *reactive action* $cnd \triangleright \overline{mnf}$. The semantics of $\mathcal{O}bs$ is defined by the rules:

[3]In [14], where the framework is extended to agent collaboration as well, v is supposed to be a conditioning for another agent, but here this constraint is released for the sake of generality.

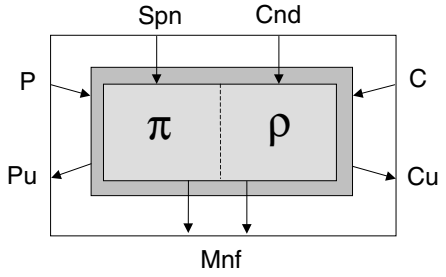

Figure 2: Core Engine

$$\frac{p_0 \xrightarrow{spn}_{\mathcal{S}} p' \qquad \langle p', c_0, spn \rangle \lhd \rhd_{\pi} \langle pu, cu, \overline{mnf} \rangle \qquad p' \xrightarrow{pu}_{\mathcal{P}u} p \qquad c_0 \xrightarrow{cu}_{\mathcal{C}u} c}{p_0 \lceil c_0 \rfloor \xrightarrow{spn \diamond \overline{mnf}}_{\mathcal{O}bs} p \lceil c \rfloor}$$

$$\frac{c_0 \xrightarrow{cnd}_{\mathcal{C}} c' \qquad \langle p_0, c', cnd \rangle \lhd \rhd_{\rho} \langle pu, cu, \overline{mnf} \rangle \qquad p_0 \xrightarrow{pu}_{\mathcal{P}u} p \qquad c' \xrightarrow{cu}_{\mathcal{C}u} c}{p_0 \lceil c_0 \rfloor \xrightarrow{cnd \triangleright \overline{mnf}}_{\mathcal{O}bs} p \lceil c \rfloor}$$

In the former case, a transition

$$p_0 \lceil c_0 \rfloor \xrightarrow{spn \diamond \overline{mnf}}_{\mathcal{O}bs} p \lceil c \rfloor$$

makes the agent in position $p_0 \lceil c_0 \rfloor$ move to position $p \lceil c \rfloor$ producing the manifestations $\overline{mnf}$. In particular, as the place changes due to the spontaneous move spn, relation π provides (i) a place update leading to p, (ii) a configuration update leading to c, and (iii) a set of manifestations $\overline{mnf}$. In the latter case, a transition

$$p_0 \lceil c_0 \rfloor \xrightarrow{cnd \triangleright \overline{mnf}}_{\mathcal{O}bs} p \lceil c \rfloor$$

makes the agent in position $p_0 \lceil c_0 \rfloor$ receive the conditioning cnd, thus moving to position $p \lceil c \rfloor$ producing the manifestations $\overline{mnf}$. In particular, as the configuration is conditioned by cnd, relation ρ provides (i) a place update leading to p, (ii) a configuration update leading to c, and (iii) a set of manifestations $\overline{mnf}$.

Some details of the core engine, which is formally described by transition systems $\mathcal{O}bs$, are pictorially given in Figure 2. The processing of observation actions is fired by either a spontaneous move (Spn) or a conditioning (Cnd), respectively leading to a proactive behaviour (π) or a reactive behaviour (ρ). In both cases the current place (P) and configuration (C) are processed producing a place update (Pu) and a configuration update (Cu), and a set of manifestations (Mnf) is sent out.

3.3 Example

In order to provide some flavours of the formal framework presented in this paper, a simple example is here described that focuses on how the observable behaviour of an agent can be generally specified, and how its evolution can be represented in terms of $\mathcal{O}bs$ transitions. Consider the case of a monitor agent in charge of maintaining some knowledge representation about the environment it lives in, modelled as a function $f \in Q \mapsto D$ where Q is the set of queries of interest and D is the set of possible results representing the data. The monitor agent can issue a query $q \in Q$ to a gatherer agent g – ranging in the set G of agents populating the environment – which is then in charge of finding

the information and replying a message. Also, the monitor agent may show some cooperative attitude, and be willing to reply to analogous queries q providing a result.

The observable behaviour of the monitor agent is given as follows. The set of places P can be defined as:

$$P = \langle \mathtt{k}{:}(Q \mapsto D), \mathtt{i}{:}(Q \times G)_\perp \rangle$$

A generic place p is then made of by two components $p_\mathtt{k} = f$ and $p_\mathtt{i} = \langle q, g \rangle$, respectively representing the knowledge function f and the agent willing to issue a request for query q to gatherer g. When no such requests are currently pending, $p_\mathtt{i}$ has value $\perp$. Spontaneous moves are terms $sIssue(q, g)$ modelling the agent deciding to request query q to gatherer g. Place updates are terms $puReset$ and $puKnowl(q, d)$, respectively modelling the place returning in the state where no gathering requests are issued, and the agent current knowledge being updated by associating query q to datum d. The semantics of spontaneous moves and place updates is defined by the simple rules:

$$p[\mathtt{i} \mapsto \perp] \xrightarrow{sIssue(q,g)}_{\mathcal{S}} p[\mathtt{i} \mapsto \langle q, g \rangle]$$

$$p[\mathtt{i} \mapsto \langle q, g \rangle] \xrightarrow{puReset}_{\mathcal{P}u} p[\mathtt{i} \mapsto \perp]$$

$$p[\mathtt{k} \mapsto f] \xrightarrow{puKnowl(q,d)}_{\mathcal{P}u} p[\mathtt{k} \mapsto f[q \mapsto d]]$$

The set of configurations C can be defined as:

$$C = \langle \mathtt{k}{:}(Q \times D)_\perp, \mathtt{i}{:}(Q \times G)_\perp \rangle$$

Given a configuration c, $c_\mathtt{k} = \langle q, d \rangle$ represents a reply to the query q with result d that is waiting to be processed, while $c_\mathtt{i} = \langle q, g \rangle$ represents an external entity g asking for the query q. In both cases, when the two components have value $\perp$ the corresponding request is not currently pending. Conditionings are terms $cRes(q, d)$ and $cQry(q, g)$, respectively representing result d to query q replied to the agent, and query q asked to the agent by g. Configuration update $cuDrop$ erases the effect of a conditioning on the configuration. The semantics of conditionings and configuration updates is defined by:

$$c[\mathtt{k} \mapsto \perp] \xrightarrow{cRes(q,d)}_{\mathcal{C}} c[\mathtt{k} \mapsto \langle q, d \rangle]$$

$$c[\mathtt{i} \mapsto \perp] \xrightarrow{cQry(q,g)}_{\mathcal{C}} c[\mathtt{i} \mapsto \langle q, g \rangle]$$

$$c[\mathtt{k} \mapsto \langle q, d \rangle] \xrightarrow{cuDrop}_{\mathcal{C}u} c[\mathtt{k} \mapsto \perp]$$

$$c[\mathtt{i} \mapsto \langle q, g \rangle] \xrightarrow{cuDrop}_{\mathcal{C}u} c[\mathtt{i} \mapsto \perp]$$

The set of manifestations sent out by the monitor agent is defined as:

$$M ::= g \uparrow request(q) \;\mid\; g \uparrow data(q, d)$$

respectively modelling a gathering request for the query q and a reply for the query q providing datum d, both sent to the external entity g.

The agent observable behaviour $\langle \pi, \rho \rangle$ is defined as follows. The agent proactive behaviour is defined so as to send gathering requests correspondingly to spontaneous moves of the kind $sIssue(q, g)$, which is expressed by the rule:

$$\langle p, c, sIssue(q, g) \rangle \lhd \rhd_\pi \langle puReset, \perp_{CU}, g \uparrow request(q) \rangle$$

The agent reactive behaviour is defined so as to handle conditionings. While $cRes(q, d)$ updates the current knowledge, $cQry(q, g)$ produces a reply providing the information d as a response to query q[4].

$$\langle p, c, cRes(q, d) \rangle \lhd \rhd_\rho \langle puKnowl(q, d), cuDrop, \bullet \rangle$$

$$\langle p, c, cQry(q, g) \rangle \lhd \rhd_\rho \langle \perp_{PU}, cuDrop, g \uparrow data(q, p_\mathtt{k}(q)) \rangle$$

With this formalisation, it is possible to represent the monitor agent interacting with the environment by means of a sequence of transitions in the transition system $\mathcal{O}bs$, that is, as a sequence of reactive and proactive transitions. Since the monitor agent handles all the pending requests in a reactive way, all the equilibrium states are characterised by the void configuration $\langle \mathtt{k}{:}\perp, \mathtt{i}{:}\perp \rangle$, which here is denoted by c_0. Consider for example the following evolution. The monitor agent has initially the knowledge represented by function $f_0 = \{q_1 \mapsto d_1\}$, and then interacts with its environment through the history: (i) the agent asks information about the query q_0 to gatherer g_0, then (ii) receives datum d_0 as a reply, and (iii) finally receives a request for the query q_1 and replies datum d_1. This evolution can be represented by the sequence of transitions:

$$\langle \mathtt{k}{:}f_0, \mathtt{i}{:}\perp \rangle \lceil c_0 \rfloor \xrightarrow{sIssue(q_0,g_0) \,\diamond\, q_0 \uparrow request(g_0)}_{\mathcal{O}bs}$$

$$\langle \mathtt{k}{:}f_0, \mathtt{i}{:}\perp \rangle \lceil c_0 \rfloor \xrightarrow{cRes(q_0,d_0) \,\rhd\, \bullet}_{\mathcal{O}bs}$$

$$\langle \mathtt{k}{:}f_0[q_0 \mapsto d_0], \mathtt{i}{:}\perp \rangle \lceil c_0 \rfloor \xrightarrow{cQry(q_1,g_0) \,\rhd\, g_0 \uparrow data(q_1,d_1)}_{\mathcal{O}bs}$$

$$\langle \mathtt{k}{:}f_0[q_0 \mapsto d_0], \mathtt{i}{:}\perp \rangle \lceil c_0 \rfloor \xrightarrow{\cdots}_{\mathcal{O}bs} \ldots$$

3.4 A specification language

We believe that this formal framework can be the basis for a full engineering methodology for agents, based on the concept of observation. As a further step towards studying such a methodology, based on this formal framework in [15] we developed a language for specifying an agent observable behaviour. This is meant to provide a more natural syntax and a structured specification, borrowing some techniques from logic-based and object oriented programming languages. For instance, according to this language the specification of the agent behaviour expressed in previous example is shown in Figure 3.

Instead of relying on a mathematical specification, by this tool an agent behaviour can be directly specified using a programming language-like syntax. The specification include: conceptual inputs and outputs of the agent core (`signature`), shape of the agent core (`structure`), and reactive and proactive agent behaviour (`dynamics`). Also, an external source of predicates can be specified that implements low-level functionalities (`import`), and global variables can be defined along with constraints (`constraints`).

The reader interested in more insights about this language is forwarded to [15], where its syntax, semantics, and pragmatics are studied in detail.

4. APPLICATIONS

In this section a number of frequently-occurring interaction patterns for agents are analised, showing how they can be modelled using the observation framework. In particular,

[4]For simplicity, here we supposed the agent is able to provide a response for any query q.

```
agent monitor{

 import predicates { isfun/1,funupd/4,funval/3 }

 constraints { isfun(F),isfun(F1) }

 signature{
  conditionings     {c_res(Q,D);c_query(Q,G);}
  spontaneous moves {s_issue(Q,G);}
  manifestations    {msg(G,request(Q));msg(G,data(Q,D));}
 }
 structure{
  place{
   state {p(F,Ip)} initial [p(null,null)] updates{
    pu_reset: FROM [p(F,Ip)] TO [p(F,null)];
    s_issue(Q,G): FROM [p(F,null)] TO [p(F,ip(Q,G))];
    pu_knowl(Q,D):
            FROM [p(F,Ip)] TO [p(F1,Ip)] IF funupd(F,Q,D,F1)
   }
   configuration {c(K,Ic)} initial [c(null,null)] updates{
    cu_drop: FROM [c(K,ic(Q,G))] TO [c(K,null)];
    cu_drop: FROM [c(kc(Q,D),Ic)] TO [c(null,Ic)];
    c_res(Q,D): FROM [c(null,Ic)] TO [c(kc(Q,D),Ic)];
    c_query(Q,G): FROM [c(K,null)] TO [c(K,ic(Q,G))];
   }
 }
 dynamics{
  proactiveness{
   s_issue(Q,G) in [P,C]: fires
     <PU pu_reset,CU null,MNF msg(G,request(Q))>
  }
  reactiveness{
   c_res(Q,D) in [P,C]: fires
     <PU pu_knowl(Q,D),CU cu_drop,MAN null>
   c_query(Q,G) in [p(F,null),C]: fires
     <PU null,CU cu_drop,MAN msg(G,data(Q,D))> IF funval(F,Q,D)
}}}
```

Figure 3: Specification of the monitor agent

for each different interaction pattern a structure of the kind

$$P = \langle \alpha_1 : R_1, \ldots \alpha_n : R_n, \ldots \rangle$$
$$C = \langle \beta_1 : S_1, \ldots \beta_m : S_m, \ldots \rangle$$

is assumed for places and configurations. The spontaneous moves, place updates, conditionings, and configuration updates that involve that interaction pattern are built so as to have effect only on components p_{α_1},..,p_{α_n} and c_{β_1},..c_{β_m} of place and configuration. As a result, the details of the management of the various interaction patterns are analised separately, and the examples provided here can be directly exploited for the specification of similar aspects in multi-agent applications.

4.1 Reactive replies

Very often agents maintain some knowledge, and may be willing to provide information on it to the other agents of the multi-agent system (MAS). In one feasible interaction pattern for sharing this knowledge, the requestor sends a message specifying the query $q \in Q$, and the agent reactively replies a result $d \in D$, that is, processing the request as soon as it arrives. This pattern is appropriate when the query result is already available, or generally when the process of obtaining it can be modelled through a function $Q \mapsto D$ computed at the time the request arrives.

This kind of interaction pattern can be basically modelled through a conditioning whose evaluation produces a manifestation and no updates on the place. Place and configuration can be modelled as records of the kind

$$P = \langle \gamma : (Q \mapsto D), \ldots \rangle \qquad C = \langle \gamma : (Q \times O)_\perp, \ldots \rangle$$

so that for any position $p \lceil c \rfloor$, p_γ represents the knowledge function, and c_γ is used to access either (i) the pair $\langle q, o \rangle$, which represents entity o asking for query q, or (ii) $\perp$ if none request currently occurs.

Spontaneous moves and place updates are not affected by the management of reactive replies. Instead, the conditioning $cGet(q)^o$ and the configuration updates $cuDrop$ are introduced that respectively add and drop a request in the configuration as follows:

$$c \xrightarrow{cGet(q)^o}_{\mathcal{C}} c[\gamma \mapsto \langle q, o \rangle] \qquad c \xrightarrow{cuDrop}_{\mathcal{C}u} c[\gamma \mapsto \perp]$$

Finally, a reactive behaviour rule should be included in the specification of ρ that produces manifestations in response to conditionings $cGet(q)^o$:

$$\langle p, c, cGet(q)^o \rangle \lhd \rhd_\rho \langle \perp_{PU}, cuDrop, o \uparrow p_\gamma(q) \rangle$$

Notice that this interaction pattern has been already exploited in the example of Section 3.3 to model the monitor agent receiving query requests.

In general, reactive replies could have been modelled using a simpler formalisation, where conditionings do not change configuration at all, but simply work as triggers for the evaluation. In this way, it would be possible to avoid the specification of any configuration update. However, the formalisation provided here seems to better fit the ontology for observation, which defines conditionings as changes in the configuration caused by coordinators.

4.2 Conditioned replies

A more powerful and useful interaction pattern for agents is the one where the reply to a request is sent only when a given condition is satisfied in the agent, or generally, when an event occurs on it. Here, a request is supposed to specify the query $q \in Q$ as well as the condition $k \in K$ to be verified. Beside a function in $Q \mapsto D$ – providing the result d for a query q as in the case of reactive replies –, the agent place is also supposed to be equipped by a function $eval \in K \mapsto \mathbb{B}$ evaluating conditions to boolean values. As a result, an event occurring in the agent is modelled as the evaluation of a given condition changing from `false` to `true`. Places and configurations are now of the kind:

$$P = \langle \kappa : (Q \mapsto D), \epsilon : (K \mapsto \mathbb{B}), \ldots \rangle$$
$$C = \langle \kappa : (\overline{Q \times K \times O}), \ldots \rangle$$

Given a position $p \lceil c \rfloor$, p_κ represents the current knowledge function, p_ϵ the function evaluating conditions, and c_κ a multiset of tuples of the kind $\langle q, k, o \rangle$ – each representing a pending request for a conditioned reply, specifying query q, condition k, and observer o.

Spontaneous moves of the kind $sCond(K')$ are introduced that represent the subset of conditions $K' \subseteq K$ whose evaluation changed from `false` to `true`. Their semantics is defined by the rule:

$$\frac{k \notin K' \Rightarrow eval(k) = eval'(k) \qquad k \in K' \Rightarrow eval(k) = \texttt{false} \wedge eval'(k) = \texttt{true}}{p[\epsilon \mapsto eval] \xrightarrow{sCond(K')}_{\mathcal{S}} p[\epsilon \mapsto eval']}$$

The conditioning $cGetk(q,k)^o$ represents the request for query q, with condition k, performed by entity o. The configuration update $cuDrop(q,k)^o$ erases the effect of condi-

tioning $cGetk(q,k)^o$, while $cuDrop(K')$ models erasing the effects of all requests waiting for some $k \in K'$.

$$c[\kappa \mapsto \overline{\langle q,k,o\rangle}] \xrightarrow{cGetk(q,k)^o}_{\mathcal{C}} c[\kappa \mapsto \langle q,k,o\rangle | \overline{\langle q,k,o\rangle}]$$

$$c[\kappa \mapsto \langle q,k,o\rangle | \overline{\langle q,k,o\rangle}] \xrightarrow{cuDrop(q,k)^o}_{\mathcal{C}u} c[\kappa \mapsto \overline{\langle q,k,o\rangle}]$$

$$\frac{\begin{array}{c}\langle q_0,k_0,o_0\rangle \in \overline{\langle q',k',o'\rangle} \Rightarrow k_0 \in K' \\ \langle q_0,k_0,o_0\rangle \in \overline{\langle q,k,o\rangle} \Rightarrow k_0 \notin K'\end{array}}{c[\kappa \mapsto \overline{\langle q',k',o'\rangle} | \overline{\langle q,k,o\rangle}] \xrightarrow{cuDrop(K')}_{\mathcal{C}u} c[\kappa \mapsto \overline{\langle q,k,o\rangle}]}$$

Finally, the observable behaviour rules are as follows. When the conditioning $cGetk(q,k)^o$ reaches the agent, if the condition k is evaluated to true the manifestation is produced and the configuration is left unchanged. Otherwise, the configuration keeps track of it, and when a spontaneous move $sCond(K')$ occurs all the pending requests waiting for some $k \in K'$ are served. This is obtained by adding the following reactiveness and proactiveness rules:

$$\frac{p_\epsilon(k) = \texttt{false}}{\langle p,c,cGetk(q,k)^o\rangle \lhd \rhd_\rho \langle \bot_{PU}, \bot_{CU}, \bullet\rangle}$$

$$\frac{p_\epsilon(k) = \texttt{true}}{\langle p,c,cGetk(q,k)^o\rangle \lhd \rhd_\rho \langle \bot_{PU}, cuDrop(q,k)^o, o \uparrow p_\kappa(q)\rangle}$$

$$\frac{\overline{o \uparrow d} = \{o \uparrow p_\kappa(q) : getw(q,k)^o \in c_\kappa \wedge k \in K'\}_M}{\langle p,c,sCond(K')\rangle \lhd \rhd_\pi \langle \bot_{PU}, cuDrop(K'), \overline{o \uparrow d}\rangle}$$

The example modelled here also generalises the cases where the event triggering the manifestation is not explicitly specified by the request, but is somehow implicitly related to the query. For instance, this happens when the manifestation has to be sent the first time the query result changes, or when the conditioning fires a computational process whose result produces the manifestation. However, in these cases the formalisation is mostly similar to the one presented above, so here its detailed description is avoided for brevity.

4.3 Subscribing

Instead of monitoring the current value of some information by continuously sending reactive replies, or re-iterating conditioned replies, it may be more effective and efficient for an agent to rely on a notification-based interaction protocol. In particular, the requestor may send a request registering its interest in receiving a notification each time the value of a certain knowledge item changes. To this end, set of places and configurations are defined as:

$$P = \langle \chi : (Q \mapsto D), \ldots\rangle \qquad C = \langle \chi : (\overline{Q \times O}), \ldots\rangle$$

Considering a generic position $p\lceil c \rfloor$, p_χ denotes the knowledge function as usual, while c_χ denotes a multiset of tuples of the kind $\langle q,o\rangle$ – representing a subscribing request for query q performed by observer o. Conditioning $cGets(q)^o$ represents entity o registering its interest in changes to the result of query q, with semantics:

$$c[\chi \mapsto \overline{\langle q,o\rangle}] \xrightarrow{cGets(q)^o}_{\mathcal{C}} c[\chi \mapsto \langle q,o\rangle | \overline{\langle q,o\rangle}]$$

Queries $q \in Q' \subseteq Q$ changing their result are modelled through the spontaneous move $sChg(Q')$, with semantics:

$$\frac{q \in Q' \Leftrightarrow get(q) \neq get'(q)}{p[\chi \mapsto get] \xrightarrow{sChg(Q')}_{\mathcal{S}} p[\chi \mapsto get']}$$

The observable behaviour rule is defined as the agent accepting a conditioning, correspondingly sending the current result, and later proactively notifying each change on that value:

$$\langle p,c,cGets(q)^o\rangle \lhd \rhd_\rho \langle \bot_{PU}, \bot_{CU}, o \uparrow p_\chi(q)\rangle$$

$$\frac{\overline{o \uparrow d} = \{o \uparrow p_\chi(q) : \langle q,o\rangle \in c_\chi \wedge q \in Q'\}_M}{\langle p,c,sChg(Q')\rangle \lhd \rhd_\pi \langle \bot_{PU}, \bot_{CU}, \overline{o \uparrow d}\rangle}$$

Adapting the conditioned replies to a notification form can be done in a similar way.

4.4 Request Interruption

While in general reactive replies do not sensibly affect the agent behaviour, both conditioned replies and subscriptions may result in a considerable burden for the agent, which should keep track of them and continuously evaluate firing conditions. Our formalisation emphasises this aspect by means of agent configuration: reactive replies remains in the configuration only temporarily during motion, while conditioned replies and subscriptions affect the configuration until some condition is satisfied – in the former case – or undefinitely – in the latter case. In a cooperative MAS, when an agent is no more willing to receive the result of a previously sent request, it may communicate its intention of aborting it. Consequently, the receiving agent can release resources that are processing this request.

Consider for simplicity the case of subscribing requests, and suppose a requestor o is willing to interrupt a subscription for query q through a conditioning $cTer(q)^o$. To this end, the configuration is modified so as to include a termination request as well:

$$C = \langle \chi : (\overline{Q \times O}), \tau : (Q \times O)_\bot, \ldots\rangle$$

which is represented by component $c_\tau = \langle q,o\rangle$. The semantics for conditioning $cTer(q)^o$ is:

$$c \xrightarrow{cTer(q)^o}_{\mathcal{S}} c[\tau \mapsto \langle q,o\rangle]$$

The effect of a conditioning $cTer(q)^o$ is to drop both the pending request and the termination request from the configuration, which is modelled by configuration update $cuTer(q)^o$ as follows:

$$c[\chi \mapsto \langle q,o\rangle | \overline{\langle q,o\rangle}] \xrightarrow{cuTer(q)^o}_{\mathcal{C}u} c[\chi \mapsto \overline{\langle q,o\rangle}][\tau \mapsto \bot]$$

$$\frac{\langle q,o\rangle \notin \overline{\langle q,o\rangle}}{c[\chi \mapsto \overline{\langle q,o\rangle}] \xrightarrow{cuTer(q)^o}_{\mathcal{C}u} c[\chi \mapsto \overline{\langle q,o\rangle}][\tau \mapsto \bot]}$$

The corresponding reactive rule is simply defined as:

$$\langle p,c,cTer(q)^o\rangle \lhd \rhd_\rho \langle \bot_{PU}, cuTer(q)^o, \bullet\rangle$$

4.5 Informing

All the interaction patterns considered above mainly concern the agent receiving and processing knowledge queries. In particular, none of these interaction patterns has any effect on the agent place, as emphasised by the fact that no place updates were specified. However, it may happen that

the environment alters the agent state, for instance when it informs the agent about some fact.

Set F is used as the set of facts an agent can be informed of. The set of places and configurations can be defined as:

$$P = \langle \iota : \overline{F}, \ldots \rangle \qquad C = \langle \iota : F_{\perp}, \ldots \rangle$$

so that given a position $p \lceil c \rfloor$, p_ι is the multiset of facts the agent has been informed of, while c_ι represents either a fact currently conditioned or $\perp_F$ – representing none fact currently occurring in the configuration. Conditioning $cInf(f)$ and configuration update $cuInf$ are simply defined as:

$$c \xrightarrow{cInf(f)}_{\mathcal{C}} c[\iota \mapsto f] \qquad c \xrightarrow{cuInf}_{\mathcal{C}u} c[\iota \mapsto \perp_F]$$

The agent place is affected by an inform message through the place update $puInf(f)$, while processing the received facts is modelled through the spontaneous move $sInf$, which removes facts from the place:

$$p[\iota \mapsto \overline{f}] \xrightarrow{puInf(f)}_{\mathcal{P}u} p[\iota \mapsto f|\overline{f}] \qquad p \xrightarrow{sInf}_{\mathcal{S}} p[\iota \mapsto \bullet]$$

The observable behaviour now is defined by the two rules:

$$\langle p, c, cInf(f) \rangle \lhd \rhd_\rho \langle puInf(f), cuInf, \bullet \rangle$$

$$\langle p, c, sInf \rangle \lhd \rhd_\pi \langle \perp_{PU}, \perp_{CU}, \bullet \rangle$$

Notice that this pattern is somehow similar to the one exploited in Section 3.3 to model the monitor agent receiving new knowledge from outside.

4.6 Proactive sending

Whereas all the previous interaction patterns describe the effect of a message on the receiving agent, the formalism presented in this paper is also able to model both the causes and the effects of an agent proactively deciding to send a message. In its simpler version, this behaviour can be represented through a spontaneous move $sSnd(m)$ – where m ranges over the set of messages M – affecting places of the kind

$$P = \langle \sigma : M_{\perp}, \ldots \rangle$$

and through a place update $puSnd$ erasing its effects, defined by:

$$p \xrightarrow{sSnd(m)}_{\mathcal{S}} p[\sigma \mapsto m] \qquad p \xrightarrow{puSnd}_{\mathcal{P}u} p[\sigma \mapsto \perp_M]$$

and with the proactiveness rule

$$\langle p, c, sSnd(m) \rangle \lhd \rhd_\pi \langle puSnd, \perp_{CU}, m \rangle$$

In this kind of formalisation, all the burden of packing a manifestation message is assigned to the agent internal machinery, and the spontaneous move directly provides the message to be sent. In general, it may be more appropriate to keep track of the proactive behaviour within the configuration state, so that while a spontaneous move models the intention of a manifestation, other details on the message to be sent are prepared according to the configuration content.

Suppose a given event occurring in the agent should produce a manifestation to an observer, and that the manifestation has to contain information about the agent current state. This can be modelled by putting in the configuration a sort of *internal observer* [13], that is, an item responsible for keeping track of (i) the firing condition k, (ii) the query q of interest, and (iii) the manifestation observer. When a spontaneous move occurs that represents condition k evaluating from `false` to `true`, relation π accesses the information on the internal observer within the configuration, producing the manifestation.

The reader should find a deep similarity between such one internal observer and a pending request for a conditioned reply. Two basic differences exist. On the one hand, an internal observer of this kind is likely to occur in the configuration indefinitely, or at least, until a request for termination is processed. From this point of view, the techniques for condition replies and subscriptions can be mixed, by leaving the internal observer in the configuration indefinitely – or until it is terminated. On the other hand, while the conditioned reply is inserted in the configuration by means of a conditioning, the internal observer is more likely to be present there from the agent startup, or also inserted there due to a spontaneous move modelling the fact that the agent is willing to start manifesting a given proactive behaviour. As a result, the corresponding modellisation is mostly similar to that shown in Section 4.2, which has been exploited in the monitor agent example to model the agent issuing a request to a gatherer.

5. RELATED WORKS

In this work we developed a modelling technique for agent-based systems based on the idea of representing agents as observable sources. While this model follows the framework presented in [14], the operational semantics presented here has been developed so as to ease the task of providing expressive specifications. In particular, here both place and configuration are described in a more abstract way in terms of transition systems, where spontaneous moves and conditionings are seen as actions. Moreover, instead of relying on the concepts of *configuration-atoms*, and on their selection and evaluation process as in [14], here we defined core engine by reactiveness and proactiveness relations, which we believe could represent typical behaviour of agents in a more natural way.

Two main applications have been studied up now. Firstly, in [15] we showed how a language can be defined for the specification of an agent interactive behaviour that allows for a semantic definition in terms of the observation framework. In Section 3.4 we provided a simple application example which can be directly obtained from the usual definition "by rules" of transition systems, yet resulting in a more intuitive description

Secondly, in [16] we analised the applicability of our framework to the semantics of agent communication languages (ACL). In fact, the observable source model indeed describes the effects of receiving messages, as well as which internal behaviour is responsible for sending messages. So, this can be interpreted as providing preconditions and effects of agent communicative acts, following e.g. the approach of FIPA to the definition of its ACL semantics [5]. In particular, this is based on the concept of agent mental state, with *feasible preconditions* and *rational effects* being defined in terms of a multi-modal language (called SL), with operators for beliefs, desire, intentions, and uncertain beliefs – namely, based on the BDI framework [4]. Instead, our observation framework may allow an ACL semantics to be formalised abstracting from the concept of mental state, by means of an operational semantics [16]. Whether this approach can fulfill the

requirements issued e.g. in [11] to overcome the limits of existing ACL semantics is still unclear, and deserves further studies.

Our model of the interactive behaviour of an agent is based on transition systems as usual semantics for process algebras [2], from which some techniques have been borrowed. While in that context communications are typically either synchronous (as in CCS [9]) or asynchronous [3], we describe an agent behaviour by reactive actions and proactive actions, modelling higher-level atomic actions representing multiple input and output communicative acts.

Other approaches aim at defining architectures for software component behaviour focussing on their interactions, whose most notable example is the Actors model [1, 7]. An *actor* is defined as a purely reactive software entity, which receives messages and correspondingly changes its behaviour and send other messages. Conversely, as in most agent models the observation framework deals with agent proactiveness [17], that is, with agents featuring an inner control responsible for changing its status and sending messages.

Another approach to modelling agent communications in terms of labelled transition systems has been developed in [12]. There, a programming language is defined where agents are characterised by their mental state – including belief states and a goal state – and where agent communications follow the rendezvous schema, a version of the classic remote procedure call (RPC) where the target agent processes requests using an interleaved pattern. Then, the language is defined through an operational semantics, describing how agents evolve by internal computations (such as believes update) and through communications. There, however, the focus is put on modelling an agent mental state and its dynamics, while the goal of the observation framework is more to emphasise the agent collaborative aspects – namely, its interactive behaviour. Another relevant difference is related to the technical treatment: instead of describing agent aspects through a programming language, here a generic architecture is developed – the model of an agent as an observable source – which can be specialised to different behaviours by changing the source specification.

6. CONCLUSIONS

In this paper we developed a formal model for representing the individual agent behaviour on top of the ontological framework developed in [14], which is based on concepts related to the observation issue. In particular, we considered patterns suggested by existing ACLs: for instance, interaction patterns shown in Sections 4.1-4.5 are somehow related to particular FIPA ACL communicative acts, such as QUERY (reactive replies), QUERY-IF (conditioned replies), SUBSCRIBE (subscribing), CANCEL (request interruption), and INFORM (informing). As a result, it would be interesting to devise out a comprehensive mapping of FIPA ACL communicative acts onto the observation framework, which is left as future work.

Our experience suggests that the observation approach can be a useful tool at every level of the engineering process – not only specification, but design, implementation and validation as well. In particular, it would be interesting to evaluate whether our transition system semantics for agents could support the applicability of existing tools for designing and validating interactive systems [8].

7. REFERENCES

[1] G. Agha. *Actors: A Model for Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, Amsterdam, London, New York, Oxford, Paris, Shannon and Tokyo, 2001.

[3] G. Boudol. Asynchrony and the π-calculus. Rapport de Researche, INRIA, Sophia Antipolis, Number 1702, May 1992.

[4] P. R. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–361, 1990.

[5] FIPA. FIPA communicative act library specification. `http://www.fipa.org`, 2000. Doc. XC00037H.

[6] R. v. Glabbeek. The linear time – branching time spectrum I. The semantics of concrete, sequential processes. In Bergstra et al. [2], chapter 1, pages 3–100.

[7] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[8] M. R. J.F.Groote. Algebraic process verification. In Bergstra et al. [2], chapter 17, pages 1151–1208.

[9] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[10] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Pres, 1999.

[11] M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[12] R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Operational semantics for agent communication languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *LNAI*, pages 80–95. Springer-Verlag, 2000.

[13] M. Viroli, G. Moro, and A. Omicini. On observation as a coordination pattern: An ontology and a formal framework. In *16th ACM Symposium on Applied Computing (SAC 2001)*, pages 166–175, Las Vegas (NV), 11–14 Mar. 2001. ACM.

[14] M. Viroli and A. Omicini. Modelling agents as observable sources. *Journal of Universal Computer Science*, 8, 2002.

[15] M. Viroli and A. Omicini. A specification language for agents observable behaviour. In *2002 International Conference on Artificial Intelligence (IC-AI'02)*, Las Vegas, USA, 24–27,June 2002.

[16] M. Viroli and A. Omicini. Towards an alternative semantics for FIPA ACL. In R. Trappl, editor, *Cybernetics and Systems 2002*, volume 2, pages 689–694, Vienna, Austria, 2002. Austrian Society for Cybernetic Studies. 16th European Meeting on Cybernetics and System Research (EMCSR 2002), 2–5 Apr. 2002, Vienna, Austria, Proceedings.

[17] M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, 2000.