A Specification Language for Agents Observable Behaviour

Mirko Viroli Andrea Omicini

Abstract—One of the main issues in the engineering of multi-agent systems is to give a precise semantics to agent communicative acts, by specifying how agent interactions with the environment affect and are affected by the agent inner status – namely, by the decisions it takes and by its perception of the world. In this paper, we tackle this problem by modelling agents as observable sources, that is, as proactive and autonomous abstractions manifesting an observable status and dynamics, and allowing external entities to perceive and alter it. Based on this idea, an abstract language for specifying agents observable behaviour is presented, providing the designer with a tool for precisely characterising the relationship between agents internal aspects and agents collaborative aspects. We claim that this language can be the basis for a successful engineering methodology for agent-based systems where the concept of observation is taken as a foundational issue.

Index Terms—Agent Communication Languages, Agent Architectures, Formal Specification Languages, Agent-Oriented Software Engineering

I. INTRODUCTION

Modelling agents behaviour is a difficult challenge for software engineers, especially as far as both agent deliberation and agent collaborative aspects are concerned. On the one hand, agents are too complex abstractions to be modelled as white-box entities whose behaviour is fully known and represented, since agent inner machinery often hides cumbersome activities such as planning and inference. Also, unknowability comes in when applying agents as abstractions to tackle legacy systems, that is, wrapping those components that, independently from their complexity, are no longer completely known, but still provide reliable services.

On the other hand, modelling agents as pure black-box entities, known only in terms of the set of actions they are able to perform on the environment, is often a too severe constraint, which badly supports a true engineering methodology. While this modelling technique is quite traditional in the field of concurrency theory, being the basis of the observational equivalence approach of process algebras [1], it seems somehow inadequate to agent-based systems. In fact, autonomy makes agent interactions loosely coupled with agent inner activity, so that simply observing an agent interaction histories often provides very limited information on the actual status of the agent — its perception of the world, its goals, its intentions, and so on. For instance, an agent receiving a message provides very few information about its actual effect on the agent behaviour, for the agent can simply decide to ignore it.

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

M. Viroli and A. Omicini are with DEIS, Università degli Studi di Bologna, via Rasi e Spinelli 176, 47024 Cesena (FC), Italy – email: {mviroli,omicini}@deis.unibo.it

As a result, the most reasonable way to tackle the problem of agent modelling seems to rely on an intermediate approach, namely a grey-box approach [2], where most agent inner details are abstracted away, and only the aspects related to the agent observable behaviour are explicitly represented. In particular, this is somehow equivalent to providing an abstract architecture for agents, and describing the semantics of agent interactions by focussing on how agent dynamics results in sending and receiving communicative acts.

The most notable example of this approach is the semantics proposed by the Foundation for Intelligent Physical Agents organisation (FIPA) to its Agent Communication Language (ACL) [3]. In this proposal, any agent’s communicative act is formally described in terms of feasible preconditions (FP) that must hold in the act’s sender, and rational effects (RE) that the sender may suppose to occur on the receiver – even though these effects are not actually mandatory for the receiver, because of its need to preserve autonomy. In particular, the specification of FPs and REs is given in terms of the Semantic Language (SL) [4], a multi-modal logic language with operators for Beliefs (B), Desires (D), Intentions (I), and Uncertain Beliefs (U) based on the BDI framework [5]. As discussed in detail in [6], providing this formal semantics for communicative acts is conceptually equivalent to implicitly assuming an abstract architecture for agents taking as primary concepts the notion of agent perception of the world (beliefs and uncertain beliefs), agent goals (desires), and agent plans for achieving that goals (intentions). However, as pointed out in a number of papers [6], [7], [8], such an approach is often too specifically based on the concept of agent mental state, so that it is not widely useful to support a general description of agent communicative acts.

In fact, agents are often applied as wrappers for legacy systems, physical resources, knowledge sources, or as simple active software components participating in well-known interaction protocols. There, agents are not suitable for a BDI implementation, so that a specification based on the concept of an agent mental state is rather useless. Moreover, that specification also provides a too complex modelling tool for allowing basic engineering support, such as proving conformance of an implementation with respect to a specification [9].

In general, exploiting the grey-box approach introduces a serious challenge, calling for abstract architectures that are loosely coupled with the actual agent implementation, and that capture at the desired level of abstraction the aspects of interest of the agent observable part. So, based on our work in [6], [10], we provide an abstract architecture for agents built on the idea of modelling agents taking the
In our methodology, agents are modelled as observable sources, that is, as software components storing some knowledge and providing services for manifesting its dynamics to the environment, and for allowing the environment to affect it. This framework has been shown to suitably support the modelling of important aspects of agents [10], such as reactivity, proactiveness, social attitude, and autonomy [11], and to allow for encoding typical agent interaction schemata [12]. In general, this model seems to provide a good level of abstraction in describing the relationships between agent internal behaviour and agent collaborative behaviour, that is, between an agent’s inner behaviour and its interaction with the environment.

As a further step towards building a full engineering methodology for agent-based systems – including specification, design, implementation, and validation – in this paper we provide a language for specifying an agent observable behaviour in terms of the observable source model. This language defines signature, structure, and dynamics of the observable source abstract architecture, borrowing some features from logic languages, and providing a declarative, high-level specification. Then, we provide a semantics for this specification in terms of transition systems, by characterising agents as abstractions evolving through the interactions with the environment – namely, by providing output messages either as response to an input message, or due to the occurrence of internal events modelling proactivity.

The remainder of this paper is as follows. Section II describes in an informal way our abstract architecture for agents. Section III introduces syntax and semantics of the specification language, and Section IV discusses a simple application example. Finally, Section V provides concluding remarks.

II. Modelling Agents as Observable Sources

The reference agent model for our approach is that of an observable source [12], [6], [10]. This is based on an ontology for the observation issue in computer systems developed in [13], where a formal framework is developed that describes how software components – coming from the wide are of computer systems – allow observation of their status and its dynamics by the environment. This work inspired a modelling technique based on the concept of observation, where software components are thought of as (observable) sources cooperating with observers and coordinators. Observers are interested in perceiving manifestations of the source status and its dynamics, represented by the output messages it sends. Coordinators condition the source observable behaviour so as to set up interaction with observers, by sending input messages to the source. Interpreting agents as sources according to this ontology means to see agents in terms of the abstract architecture shown in Figure 1.

An agent is conceptually divided in two parts: the agent observable core and the agent internal machinery. On the one hand, the agent observable core is the part of the agent that is explicitly represented by the model, specifying how conditionings (Cnd) are handled and how manifestations (Mnf) are produced – that is, specifying the agent interactive behaviour. On the other hand, the agent internal machinery is the part of the agent that is abstracted away – typically involving complex aspects such as agent deliberation, planning, and inference. This division is generally necessary in order to deal with both the need of abstracting away from complex details while focusing on the interactive behaviour – which is a general practice for harnessing complexity [14], [15] –, and the need for representing causes and effects of agent interactions in terms of its internal status.

Our model goes on to the details of the observable core part dynamics, and to its relationship with both the environment and the agent internal machinery. The observable core is made of the agent place (P), representing the agent part (or abstract view) of the status affecting and affected by the environment, and the agent configuration (C), storing information on the interactions the agent is in charge of handling. Agent dynamics is either reactive, when the agent receives a conditioning, or proactive, when a spontaneous move (Spn) occurs within its place. While the former takes into account the effect of the environment on the agent, the latter is meant to represent how the agent internal machinery can affect the observable behaviour. Notice that our framework enjoys an interesting property: agent internal machinery is abstracted away, but is anyway taken into account as far as it can affect the agent observable behaviour – namely, by means of the notion of spontaneous move.

Either conditionings and spontaneous moves are processed by the agent core engine (Eng), generally producing a change on the core state, in terms of a place update (PU) and a configuration update (CU), and some manifestations to be produced and sent out to observers. So, from the point of view the of the agent interactions with the environment, this model both represents (i) the agent reactively replying to an input message (a conditioning) by sending output messages (manifestations) and by updating its state (place and configuration), and (ii) the agent proactively sending output messages and updating its state as a consequence of an internal event (a spontaneous move).
III. The Specification Language

In this section, we provide syntax and semantics of a specification language for agents observable behaviour. In short, the language is an abstract tool for formally describing the dynamics of an agent core and of the manifestations it produces as an effect of the conditionings it receives and the spontaneous moves occurring in it. Thus, semantics is given by providing a mapping between the specification and a labelled transition system representing the agent as an interactive software abstraction.

A. Syntax

The language borrows some aspects from the notation of logic languages such as Prolog. Let $\phi$ range over the finite set $F$ of (logic) functions, each with a given arity, $\nu$ over the finite set $V$ of variables, and $\tau$ over the set $T$ of terms built with functions and variables as:

$$\tau ::= \nu \mid \phi(\tau_1, \ldots, \tau_n)$$

Let $\sigma$ range over the finite set of predicate symbols, and $\psi$ over the set of predicates (atomic formulae), built applying predicate symbols to terms:

$$\psi ::= \sigma(\tau_1, \ldots, \tau_n)$$

The complete abstract syntax of our specification language is shown in Figure 2.

The specification of an agent observable core is associated to a name represented by term $\tau_N$ — which may e.g. identify the role of the agent within the Multi-Agent System (MAS) — and is mainly made by three parts: a signature, a structure, and a description of its dynamics.

The first part of the specification, the agent signature, defines the set of admissible conditionings, spontaneous moves, and manifestations for the agent. Both conditionings, spontaneous moves, and manifestations are represented in form of elements $\kappa$, that is, as collections of terms. Any element $\kappa = \tau: \psi_1, \ldots, \psi_n$ represents the set of terms matching $\tau$ and whose variables satisfy conditions $\psi_1, \ldots, \psi_n$. Applying a conditions production is always optional.

The second part of the specification is the agent structure, defining the shape of place and configuration, as well as the ways they can evolve. In particular, $\kappa_P$ represents a generic place with initial state $\tau_P^0$; and similarly, $\kappa_C$ represents the generic configuration with initial state $\tau_C^0$. The dynamics of both place and configuration — that is, the place and configuration updates along with their effect on place and configuration state — is described by a set of transition rules $transition^*$, each specifying a label $\tau_{ab}$, an old state $\tau_I$, a new state $\tau_{F}$, and a list of conditions to be satisfied, expressed by predicates $\psi_1^*, \ldots, \psi_n^*$. In the case of place specification, each transition rule means that if the conditions associated to all predicates $\psi_i^*$ are satisfied, then the place update $\tau_{ab}$ makes the place $\tau_I$ move to $\tau_F$. Configuration specification is handled similarly. Notice that we suppose conditionings to be particular cases of configuration updates, and spontaneous moves to be particular cases of place updates.

Finally, agent dynamics defines how conditionings and spontaneous moves are processed, in terms of how they affect place and configuration, and what manifestations they produce — that is, the behaviour of the core engine part. In particular, firing rules in the reactiveness part describe the effect of conditionings, while those in the proactiveness part describe the effects of spontaneous moves. Each firing rule specifies the effect of a triggering event $\tau_{trig}$ (either a conditioning or a spontaneous move, respectively) on a core in place $\tau_P$ and configuration $\tau_C$: if optional conditions $\psi_i^*$ are satisfied, then place update $\tau_{PU}$ and configuration update $\tau_{CU}$ are applied, and manifestation $\tau_{MNF}$ is sent out — conceptually representing either one ore more output messages.

A library file can also be specified (import) to import the predicate symbols used throughout the specification — namely, used to define the predicates $\psi$ appearing as conditions. Also, we suppose that the semantics of these symbols is defined by the library file, in terms of the logic theory used to check for the succeeding predicates. For instance, a library file can be a Prolog database defining the semantics of the imported predicate symbols. A library file is generally useful to support low-level functionalities, such as the management of lists, multisets, functions, and
so on.

Global conditions can be also defined (constraints) that are used to specify conditions to be satisfied throughout the whole specification, typically constraining the set of terms that can be substituted to a given variable, in any transition rule, firing rule, or \( \kappa \) definition. Their semantics is simply given by supposing that a pre-processor actually adds global constraints to predicate lists in any conditions production.

### B. Labelled Transition Systems

Labelled transition systems are a very simple, yet powerful framework to denote the operational semantics of interactive systems. As surveyed e.g. in [16], they allow for expressing a wide set of different equivalence and preorder semantics over interactive software systems. These mathematical tools, for instance, allow for deciding substitutability of two software components, and for checking conformance of a component implementation with respect to a high expressive and abstract framework for specifying interactions of software components, in this paper we define the semantics of our specification language in terms of a labelled transition system. Formally speaking, a (labelled) transition system [16] is a triple \( (S, \rightarrow, A) \), where \( S \) is said to be the set of states, \( A \) is the set of actions, and \( \rightarrow \subseteq S \times A \times S \) is a transition relation, writing \( s \xrightarrow{a} s' \) when \( \langle s, a, s' \rangle \in \rightarrow \).

The common interpretation of this mathematical structure is as a specification of the interactions between a software component and its environment. While \( S \) is the set of process states, or rather of the states of the software component of interest, \( A \) is the set of actions the process is able to perform on the environment, e.g. representing either receiving or sending messages. Correspondingly, symbol \( s \xrightarrow{a} s' \) means that the process in state \( s \) may move to state \( s' \) by performing action \( a \). Notice that this framework is intrinsically non-deterministic: (i) when both \( s \xrightarrow{a} s' \) and \( s \xrightarrow{b} s'' \) hold, either actions \( a \) or \( b \) can be performed from state \( s \), (ii) when \( s \xrightarrow{a} s' \) and \( s \xrightarrow{a'} s'' \) hold, by performing \( a \) from state \( s \) the system may move either to \( s' \) or to \( s'' \).

In order to define the content of a transition relation so as to define the semantics of the transition system, it is usual practice to specify one or more rules of the kind

\[
\begin{align*}
\text{condition} \\
\sigma \xrightarrow{\alpha} \sigma' 
\end{align*}
\]

where \( \sigma \) and \( \sigma' \) are formulae denoting elements in \( S \), \( \alpha \) is a formula denoting an action in \( A \), and condition is a predicate on the free variables of \( \sigma \), \( \sigma' \), and \( \alpha \). The intended semantics of these rules is that \( s \xrightarrow{\sigma \xrightarrow{\alpha} \sigma'} \) holds if and only if there is one variable substitution to be applied to one rule so that (i) its condition is satisfied and (ii) its formulae \( \sigma \), \( \sigma' \), and \( \alpha \) are bounded respectively to \( s \), \( s' \), and \( a \). When no condition has to be specified, only the downside part of the rule is written, meaning that any substitution of the free variables can be applied.

### C. Semantics

Given a set \( X \) we generally let meta-variable \( x \) range over \( X \), and \( x \perp \) over the set \( X = X \cup \{ \bot \} \), where \( \bot \) (or \( \perp \) for short) is typically considered an exception value over \( X \). Given a variable-to-term substitution \( \theta \in \Theta = \mathcal{V} \rightarrow \mathcal{T} \), its application to a term \( \tau \) or predicate \( \psi \) causing substitution of their variables – is denoted by symbols \( \tau \theta \) and \( \psi \theta \). In order to provide semantics to a specification, we take as reference the symbols used in Figure 2: for instance, we generally denote by symbol \( \kappa \) the definition of place.

Then, when it is useful to denote a term or predicate into \( i \)th transition or firing rule in a production, this is tagged by symbols \( P \xrightarrow{i} \) (for place update transitions), \( C \xrightarrow{i} \) (for configuration update transitions), \( \pi \xrightarrow{i} \) (for proactive firing rules), and \( \rho \xrightarrow{i} \) (for reactive firing rules) – e.g., term \( \phi_{lab} \) denoting the label into \( i \)th transition of place update specification is written \( \phi_{lab}^{P,i} \).

In order to define the semantics of our language, we suppose that the (imported) logic theory that gives semantics to predicate symbols is always able to decide about succeeding predicates in finite time. Under this assumption, we denote by \( \models \psi \) the fact that checking predicate \( \psi \) succeeds.

First of all, we associate to any element \( \kappa \) of the specification the set \( |\kappa| \) of matching terms satisfying its conditions, defined as:

\[
[\tau] := \psi_w^1, \ldots, \psi_w^n = \{ \tau \theta : \forall k, \theta \models \psi_w^k \theta \}
\]

By this notion, we define \( P = |\kappa_P| \) as the set of places, \( C = |\kappa_C| \) as the set of configurations, \( Spn = \bigcup_j |\kappa_{Spn}^j| \) as the set of spontaneous moves, \( Cnd = \bigcup_i |\kappa_{Cnd}^i| \) as the set of conditionings, and \( Mnf = \bigcup_i |\kappa_{Mnf}^i| \) as the set of manifestations. Place updates \( PU \) and configuration updates \( CU \) are defined as the sets of terms matching labels \( \tau_{lab} \) and satisfying predicates \( \psi_w \) in transition rules for place and configuration, respectively. Formally:

\[
PU = \{ \phi_{lab}^{P,i} \theta : \forall k, \theta, i \models \psi_w^{P,i} \theta \}
\]

\[
CU = \{ \phi_{lab}^{C,i} \theta : \forall k, \theta, i \models \psi_w^{C,i} \theta \}
\]

Transitions for place updates and configurations updates are given a semantics in terms of transition systems \( \{ P, \xrightarrow{p}, PU \} \) and \( \{ C, \xrightarrow{c}, CU \} \), that is, they define the content of relations \( \xrightarrow{p} \subseteq P \times PU \times P \) and \( \xrightarrow{c} \subseteq C \times CU \times C \). For instance, notations \( P \xrightarrow{pu} P' \) means that place may move from \( P \) to \( P' \) when place update \( pu \) occurs. In particular, relation \( \xrightarrow{p} \) is formally defined by the two rules:

\[
\frac{\forall k \models \psi_w^{P,i} \theta}{\tau_f \xrightarrow{P,i} \theta \xrightarrow{pu} \tau_{lab} \theta \xrightarrow{i} \tau \xrightarrow{null} \tau \}
\]

where \( null \) is a special constant (a 0-ary logic function) automatically defining trivial transitions moving a term
to itself. Analogously, for configuration updates we have:

\[ \forall k \models \psi^C_{\text{C}>\theta} \quad \tau_1 \begin{array}{c} \text{null} \\ \tau \end{array} \quad \begin{array}{c} \tau_i \end{array}C \tau \]

As already mentioned, we also suppose by construction that \( Spn \subseteq PU \) and \( Cnd \subseteq CU \), so that transitions for place and configuration also give a semantics to spontaneous moves and conditionings. In order to avoid unnecessary specification, we also assume that place updates and configuration updates default to the trivial semantics when their effect is not specified in the agent structure, that is:

\[ \exists \theta, i : \tau_{\text{lab}}^{p>\theta} = pu \quad \exists \theta, i : \tau_{\text{lab}}^{c>\theta} = cu \]

In a mostly similar way, firing rules are given a semantics in terms of definitions:

\[ \Box_{\text{p}} \subseteq (Cnd \times P \times C) \times (PU \times CU \times Mnf) \]
\[ \Box_{\text{p}} \subseteq (Spn \times P \times C) \times (PU \times CU \times Mnf) \]

and by means of rules:

\[ \forall k \models \psi_k^{C, \pi>\theta} \quad \tau_1 \begin{array}{c} \text{null} \\ \tau \end{array} \quad \begin{array}{c} \tau_i \end{array}C \tau \]

where special term null occurring as either place update, configuration update, or manifestation is given interpretation \( \perp \). Analogously to the case of place and configuration updates, we suppose that spontaneous moves and conditionings default to the trivial evaluation as follows:

\[ \exists \theta, i : \tau_{\text{trig}}^{p>\theta} = spn \quad \exists \theta, i : \tau_{\text{trig}}^{c>\theta} = cnd \]

Now, the semantics of the whole specification language is defined by an association between an agent specification and a transition system:

\[ \langle P \times C, \rightarrow_{\text{Obs}}, (Cnd \times Mnf_{\perp}) \cup (Spn \times Mnf_{\perp}) \rangle \]

The state of an agent at any time is modelled as a couple \( (p, c) \) with \( p \) current place (state) and \( c \) current configuration (state). Actions are of two kinds: reactive actions \( \langle cnd, mnf_{\perp} \rangle \) denoted by symbol \( cnd > mnf_{\perp} \), and proactive action \( \langle spn, mnf_{\perp} \rangle \) denoted by \( spn \circ mnf_{\perp} \). Notation

\[ (p, c) \xrightarrow{cnd > mnf_{\perp}} \text{Obs} (p', c') \]

means that the agent with place \( p \) and configuration \( c \) receives conditioning \( cnd \), moves to place \( p' \) and configuration \( c' \), and sends manifestations \( mnf_{\perp} \) (or nothing if this is \( \perp \)). In the same way, notation

\[ (p, c) \xrightarrow{spn \circ mnf_{\perp}} \text{Obs} (p', c') \]

means that the agent with place \( p \) and configuration \( c \) performs spontaneous move \( spn \), moves to place \( p' \) and configuration \( c' \), and sends manifestations \( mnf_{\perp} \). In particular, relation \( \rightarrow_{\text{Obs}} \) is defined by rules:

\[ (p_0, c_0) \xrightarrow{spn \circ mnf_{\perp}} \text{Obs} (p, c) \]
\[ (p_0, c_0) \xrightarrow{cnd > mnf_{\perp}} \text{Obs} (p, c) \]

In the former case, a spontaneous move \( spn \) occurs in the agent which updates the place from \( p_0 \) to \( p' \) and which is intercepted by the core engine, producing (i) a place update \( cnd \) of place \( c \) moving from \( c_0 \) to \( c' \), (ii) a configuration update \( mnf_{\perp} \) moving configuration from \( c' \) to \( c \), and manifestation \( mnf_{\perp} \) to be sent out. Analogously, the latter rule handles the case of a conditioning.

The specification in Figure 2 also defines the initial state for the agent \( \langle \tau_0^{p}, \tau_0^{c} \rangle \in P \times C \), which has to be considered as initial state of the transition system \( \rightarrow_{\text{Obs}} \).

D. Specification of an agent observable behaviour

We summarise here the logic steps that an user should follow so as to provide an useful specification of the observable behaviour of an agent of interest. Not surprisingly, we will find many similarities with common concepts of object-oriented languages, so that the definition of an agent abstract architecture can somehow be thought of as an extension to the notion of an abstract class of object-oriented languages.

The specification of an agent, as shown in Figure 2, should actually define the observable behaviour of a class \( \tau_N \) of agents, each adhering to the same kind of observable behaviour. Following the traditional terminology, a specification can be easily associated to the notion of agent role.

First of all, the predicates defining the low-level functionalities used in the specification should be imported from a library file. Importing an external logic theory allows us to separate two important concerns: the aspects related to the interactive behaviour of the agent, and the management of data values (and their types). While the former issue is the one our model is meant to focus on, the latter can be abstracted away by the idea of defining predicate symbols semantics outside the specification.
Then, the global variables along with their global constraints have to be specified. The definition of global constraints provides a list of predicates that should be satisfied throughout the specification, so as to constrain the usage of the set of variables they are built over. These variables can be considered as global variables of the specification, with the global constraints defining how they can be actually exploited. So, by the constraints construct the specification defines the global variables along with their intended type.

The next step is the definition of the agent signature, providing the terms that define conditionings, spontaneous moves, and manifestations. They basically describe the boundary of the agent observable core, with conditionings and spontaneous moves interpreted as input actions, and manifestations as output actions. Agent signature can be considered as a high-level syntactic description of the agent behaviour, with no meaning associated to inputs and outputs, similarly e.g. to the declaration of an interface in object oriented languages.

Consequently, the agent structure has to be defined that specify details of the structure and dynamics of the observable core. This defines the shape of place and configuration, their initial state, and how they can evolve by place updates and configuration updates. Since spontaneous moves are subsets of place updates and conditionings are subsets of configuration updates, agent structure also gives a semantics to the conceptual inputs of the observable core, in terms of how they act on place and configuration. Agent structure represents an high level description of the parts of the agent state and its dynamics that are relevant to the aim of defining the agent observable behaviour, abstracting away from all those aspects that do not directly affect the agent interaction, accordingly to the grey-box modelling approach.

Finally, the agent dynamics part is to be defined that separately describes the agent reactive behaviour and the agent proactive behaviour. In particular, it describes the effect of the reactive inputs (conditionings) and proactive inputs (spontaneous moves) in terms of how they change the agent state, and how they lead to outputs (manifestations). Given the definition of the agent structure, involving the lowest-level details of the specification, the dynamics part concisely represent the agent interactive behaviour.

IV. Example

In order to provide some flavour and more details on the specification language presented in this paper, here we give an application example based on a simple yet paradigmatic agent behaviour. We consider an agent storing some knowledge – e.g., representing its perception of the world it inhabits – and allowing other agents to query it so as to be informed when some of its part updates, and to propose new information. In particular, we provide examples of modelling agent reactivity (as replying to queries), agent proactivity (as informing the environment when an internal event occurs), and agent autonomy (as proposals for changes possibly not taken into account). Other features and interaction patterns typically occurring in agent-based systems are analysed in [12], where their encoding into the observable source model are provided and discussed. In our model, knowledge is represented as a partial function from facts to values, external entities query for or propose the value associated to a given fact. The specification of this agent, or rather of this role, which we call believer, is shown in Figure 3.

Imported predicates basically define abstract data types and some simple operations on them, including the types for agents, values, facts, lists, and multisets. Their straightforward implementation – which could be easily provided e.g. in Prolog – is avoided here for brevity; their informal meaning is as follows. Constants represent either agents (A), values (V), or facts (F); correspondingly, predicates isA/1, isV/1, and isF/1 check for a term belonging to
one of these sets. Terms can also represent multisets (e.g., implemented as lists), with \texttt{ismset/1} checking whether a term is a multiset, and with \texttt{mssetadd(M,E,N)} being true when multiset \texttt{N} is obtained by adding element \texttt{E} to multiset \texttt{M}. Functions are also handled, with \texttt{isfun/1} checking for a term belonging to the set of functions (e.g., implemented as lists of couples \((x, f(x))\)). \texttt{funupd(F,D,R,F2)} is true when \texttt{F2} is obtained by function \texttt{F} adding association \texttt{D} \(\rightarrow\) \texttt{R}, and \texttt{funval(F,D,R)} is true when \texttt{R} is the value associated by function \texttt{F} to \texttt{D}.

Constraints assign types to some global variables, namely \texttt{A} being an agent, \texttt{F} a fact, \texttt{V} a list, \texttt{Fn} and \texttt{Fn1} lists, \texttt{Ms} and \texttt{Ms1} multisets.

The agent signature describes (reactive and proactive) inputs, and outputs of the agent. Conditioning \texttt{ask(F,A)} models an external entity \texttt{A} querying for current value associated to fact \texttt{F}, \texttt{askwhen(F,A)} models \texttt{A} willing to be informed when value associated to \texttt{F} changes, and \texttt{prop(F,V)} a proposal for associating value \texttt{V} to fact \texttt{F}. Spontaneous moves are of the kind \((i)\) \texttt{newK(F,V)}, representing the agent changing its knowledge of fact \texttt{F} to value \texttt{V}, and \((ii)\) \texttt{eval}, modelling the agent evaluating a proposal for knowledge change. Manifestations are either responses \texttt{send(A,V)} to \texttt{ask} conditions, or a multiset of them \texttt{sendMany(Ms,V)} sending value \texttt{V} to all agents in multiset \texttt{Ms}.

The agent structure provides the abstract description of the observable core and its dynamics. Places contain both the knowledge function, and information on a proposed change \texttt{pr(F,V)} that still has to be evaluated, configuration is a function from facts to multisets of agents interested in their changes. Other place updates include \texttt{add(F,V)}, inserting a proposal of knowledge change \texttt{pr(F,V)} in the place, while configuration update \texttt{remall(F)} removes all registrations for changes on the value associated to fact \texttt{F}.

The agent dynamics describes the interactive behaviour of the agent. Conditioning \texttt{ask} is reactively processed, immediately causing the reply to be manifested to the requirer. Conditioning \texttt{prop} changes the observable status (source place) of the agent, waiting to be actually considered by spontaneous move \texttt{eval}. Then, spontaneous move \texttt{newK} models a change on knowledge, possibly causing manifestations to be sent to interested observers.

Notice that in this kind of modelling, agent’s internal aspects such as evaluation of proposals and deliberation of knowledge changes are completely abstracted away – which would typically have involved the most complex formalisations - but their effect are anyway taken into account and modelled as spontaneous moves. In this way, all the possible observable behaviours of the agent can be indeed represented by the model, in spite of the simple specification provided.

V. CONCLUSIONS

In agent-based systems, agents are typically proactive and autonomous abstractions, with their own local control, perception of the world, and ability to decide what actions to perform. On the other hand, agents often have also a role in a society, that is, they represent peculiar problem-solvers in groups carrying on social tasks. Typically, this social attitude is exploited by means of interaction protocols that the agent realises with other agents in the MAS.

The specification language introduced in this paper has the goal to provide the designer with a tool for reasoning about how agent internal aspects concerning its autonomy and proactiveness influence and are influenced by those agent aspects devoted at managing the interactions with the environment. This issue is of great concern, and requires an engineering methodology where these aspects are specified in detail and exploited for specifying, designing, implementing, and validating the behaviour of MAS. We believe that the observation framework exploited in this paper can be a suitable support for developing this methodology, which will be subject of our future research efforts.

REFERENCES


