Statecharts and Agent Technology: The Past and Future

Nikolaos I. Spanoudakis(✉)

Applied Mathematics and Computers Laboratory,
School of Production Engineering and Management,
Technical University of Crete, 73100 Chania, Greece
nikos@amcl.tuc.gr

Abstract. This work aims to bring forward the intersection between the world of statecharts and that of agent technology. We begin by disambiguating the different terms related to statecharts, i.e. state machines and finite state automata/machines. Subsequently we review their impact to agent technology, mainly in the area of Agent-Oriented Software Engineering. Our findings are that multi-agent systems modeling has used and, some times, extended the language of statecharts, mainly for modeling agent interaction protocols and for coordinating the different modules of an agent. We conclude with some future directions related to the use of statecharts by the Multi-agent Systems community in the coming years.

Keywords: Agent oriented software engineering · Statecharts · Finite state machines · Agent interaction protocols · Agent control · Engineering multi-agent systems

1 Introduction

Agent Technology and Statecharts technology are two worlds that started almost at the same time, in the eighties, the first as Distributed Artificial Intelligence [30] and the second as a method for engineering complex and reactive systems [19]. Since then, a lot has happened, this work will focus in their intersection.

Agent-oriented Software Engineering (AOSE) has long used statecharts. Initially, they were employed for modeling agent interactions [27,35] but also agent plans [10,31,32]. Agent interaction modeling is mainly concerned with defining protocols that govern an interaction. Such models have also been referred to as *inter-agent control* models [10,42]. Later, statecharts were used by AOSE methodologies for the coordination of the different agent modules, and such models are also referred to as *intra-agent control* models [10,41]. Moreover, agent platforms like the popular Java Agent Development Framework (JADE [2,3]) base complex agent behaviour to the definition of state machines.

This paper aims to collect the experience of using statecharts for agent-related research and propose some future directions based on the modern development for statecharts but also the needs of the agents community.

N. Bassiliades et al. (Eds.): EUMAS 2020/AT 2020, LNAI 12520, pp. 511–528, 2020.
https://doi.org/10.1007/978-3-030-66412-1_33

Section two provides a background on statecharts and finite state machines, also trying to disambiguate these terms. Then, section three provides an overview of the use of the language of statecharts in AOSE and generally in multi-agent systems (MAS) research. Section four discusses these findings. Section five proposes several research directions, and, finally, section six concludes.

2 Background on Statecharts

Statecharts are often confused with automata [12], finite state automata, or finite state machines (FSMs [4]). Let's try to give a formalism that will aid us throughout this paper.

Definition 1. *An FSM-like statechart can be defined as a tuple (L, δ) where:*

- $L = (S, Name)$ *is a set representing the states of the statechart, and:*
  - $S \subseteq \mathbb{N}^*$
  - *Name is a mapping from nodes to their names*
- $\delta \subseteq S \times TE \times S$ *is the set of state transitions, where* TE *is a set of transition expressions*

Harel proposed statecharts for modeling software systems [20]. According to that work, statecharts are based on an activity-chart that is a hierarchical data-flow diagram, where the functional capabilities of the system are captured by activities and the data elements and signals can flow between them. The behavioral aspects of these activities (what activity, when and under what conditions it will be active) are specified in statecharts.

While in FSMs states represent different states of the world and actions take place in the transitions, in statecharts states represent activities. Actions are still possible in transition expressions, however, these are instant actions that modify variable values and generally affect the data structures of the modeled system. On the other hand, the ability of the transition expressions to allow for events makes them compatible with FSMs in the sense that the outcomes of activities can be sensed and lead to the next state of the world. Moreover, activities can occur concurrently, or can be complex, i.e. can be analyzed to more "basic" ones.

In a sense an FSM is a restricted case of a statechart, where there is no explicit activity associated with a state, there is no hierarchy of states, there are no history connectors, and, finally, the orthogonality feature is missing. Nevertheless, there are researchers that have proposed hierarchical FSMs [6,17].

In this paper, when we refer to statecharts we imply the formalism given by Harel [19,20]. Based on that we can extend Definition 1 with regard to nodes (L) [41] (see Definition 2).

Harel defines several state types. Three are the main types of states in a statechart, i.e. *OR-states*, *AND-states*, and *basic* states. OR-states have substates that are related to each other by "exclusive-or", i.e. only one can be active at any given time, and AND-states have orthogonal components that are related by "and", they are active at the same time. Basic states are those at the bottom of the state hierarchy, i.e., those that have no sub-states. The state at the highest level, i.e., the one with no parent state, is called the root. There are some more states types, such as *start* and *end*. These are nodes without activity, which exist so that execution can start and end inside OR-states. A *condition* state allows for branching a transition. *Shallow_history* and *history* allow for "remembering" the last active state in an OR-state or a whole branch of L respectively. All these auxillary state types, i.e. *start*, *end*, *basic*, *shallow_history*, *history* and *condition* are leaves of L.

Definition 2. *A Statechart can be defined as a tuple* (L, δ) *where:*

– $L = (S, \lambda, Name, Activity)$ *is an ordered rooted tree structure representing the states of the statechart, and:*
  - $S \subseteq \mathbb{N}^*$
  - *Name is a mapping from nodes to their names.*
  - $\lambda : S \to \{and, or, basic, start, end, shallow\_history, history, condition\}$, *is a mapping from the set of nodes to labels giving the type of each node.*
  - *Activity is a mapping from nodes to their algorithms in text format implementing the processes of the respective states.*

– $\delta \subseteq S \times TE \times S$ *is the set of state transitions, where* TE *is a set of transition expressions*

Each transition from one state (source) to another (target) is labeled by a Transition Expression (TE), whose general syntax is $e[c]/a$, where e is the event that triggers the transition; c is a condition that must be true in order for the transition to be taken when e occurs; and a is an action that takes place when the transition is taken. All elements of the transition expression are optional.

Moreover, there can also be compound transitions (CT), that can have more than one source or target states. We will not refer to that level of detail in this work. The scope of a transition is the lowest level OR-state that is a common ancestor of both the source and target states.

The statechart formalism also defines execution semantics. We will give a brief overview, for the details the reader is referred to Harel and Naamad [20]. The execution of a statechart is a sequence of steps. After each step we view a snapshot of the statechart. Execution starts at *start* states. When a step is taken, the events that have happened are sensed, including retrospection events (such as the entering of a state at the previous step). When the step finishes the statechart is in a valid configuration, i.e. specific *basic* states are active and the respective OR- and AND-states up to the root. Other types of states cannot be included in the configuration (e.g. the a *start* state cannot be active in a snapshot).

When a transition occurs all states in its scope are exited and the target states are entered. Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart. If one of the statecharts becomes non-active (e.g. when the activity it controls is stopped) the other charts continue to be active and that statechart enters an idle state until it is restarted.

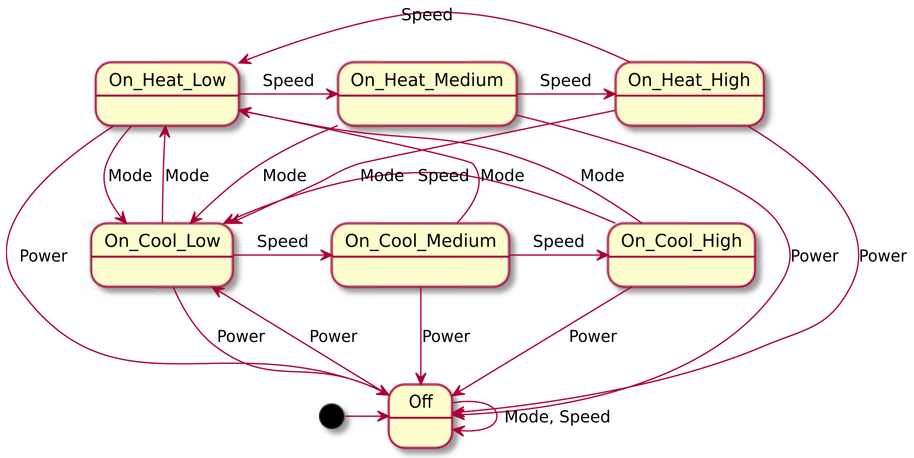

Fig. 1. FSM representation of an air conditioner. The figure was generated using the PlantText free tool, https://www.planttext.com/

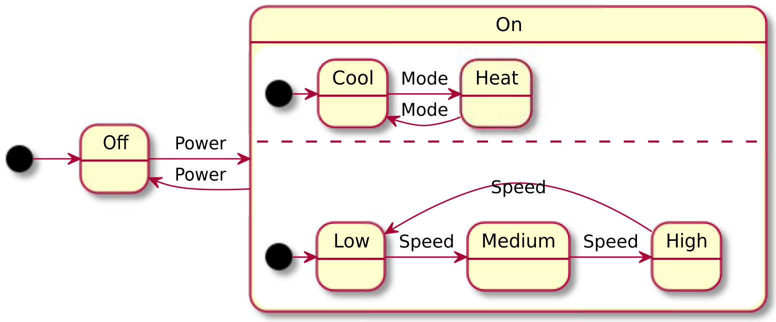

Fig. 2. Statechart representation of an air conditioner.

To illustrate the difference between a statechart and an FSM, a basic representation of an air-conditioner is provided using both formalisms, see Fig. 1 and Fig. 2. The reader will notice that using OR and AND states the statechart formalism prevents the explosion of states that takes place in FSMs as one combines contexts (in this case the context of the air-conditioner fan speed and its mode). The depicted events correspond to pressing the power, fan and mode buttons on the air-conditioner remote control.

In the Unified Modeling Language (UML), which is the mainstream language for defining object-oriented software and a standard supported by the Object Management Group (OMG), https://www.uml.org/, statecharts have been used to model the dynamic behavior of a class instance (object).

3 Statecharts and Agent Oriented Software Engineering

Statecharts were proposed by Harel for modeling reactive systems [19] and researchers in AOSE and agent interaction protocols modeling communities were quick to adopt them and propose formalisms, extentions, languages and semantics for use in agent-based systems engineering.

One of the pioneers, Moore [27], proposed an agent interactions protocol formalism based on statecharts and the Formal Language for Business Communication (FLBC), an Agent Communication Language (ACL). Usually, the message types of ACLs (or performatives) are understood as speech acts. A speech act is an act that a speaker performs when making an utterance [1]. Performatives express the intent of an agent when it sends a message to another agent. Thus, a message has four parts, a) the sender, b) the receiver, c) the performative and d) the message content (what is said). For example, the performative "inform" may be interpreted as a request that the receiving agent adds the message content to its knowledge-base. Thus, an ACL message can be defined as: $performative(sender, receiver, content)$.

For his work on conversation policies, Moore makes the assumption that developers that adopt his models can understand a formal specification and implement it in whatever way they see fit. In the FLBC, Moore defines, for example, that the message $request(sender, receiver, action)$ can express that:

a. The receiver believes that the sender wants him to do the action
b. The receiver believes that the sender wants the receiver to want to do the action

According to the work of Moore, a *conversation policy* (CP) defines a) how one or more conversation partners respond to messages they receive, b) what messages a partner expects in response to a message it sends, and, c) the rules for choosing among competing courses of action.

Moore introduces the idea of modeling the activities of the participants in a conversation as orthogonal components of a statechart. The transition expressions contain the actions of sending and receiving a message. Moore's conversation policies allow for exceptions when a conversation is interrupted by assuming that an agent has stored all allowed CPs in a kind of repository where he can browse a new policy to handle the exception in the form of a sub-dialog to the original one. When this sub-dialog terminates the original one can resume.

Statecharts were introduced in AOSE methodologies by the Multiagent Systems Engineering methodology (MaSE [9,10]). In the MaSE design phase, the first activity is about creating agent classes and then agent classes can connect to other classes indicating the possible interactions or conversations. The latter are defined in the communication class diagram, which is in the form of a finite state machine. MaSE defines a system goal oriented MAS development methodology. The authors define for the first time inter and intra-agent interactions that must be integrated.

Recognizing the fact that a protocol should have both a graphical and formal representation, Paurobally et al. [35] combined the language of statecharts and a language based on Propositional Dynamic Logic (PDL), the Agent Negotiation Meta-Language (ANML). PDL blends the ideas behind propositional logic and dynamic logic by adding actions while omitting data; hence the terms of PDL are actions and propositions. Then, the authors defined templates for transforming the ANML formulas to statecharts, extending the statecharts language in the process. The representation of all computation is in transitions, while states just describe a situation (where specific conditions hold). The representation can be general, or specialized for a specific agent participant. The expressions in the transitions are ANML formulas. The proposal of Paurobally et al. [35] and later by Dunn-Davies et al. [11] did not employ the orthogonality feature of the statecharts because they considered that the agents are not subsystems and, thus, execute on their own. If they were combined as orthogonal components for execution they would have to combine parts of interactions between temporally autonomous agents into a pseudo whole.

At the same time, König [23] presented a new possibility in inter-agent protocols definition. He used the state transition diagrams (STD) formalism to model protocols, but also decision activities, thus, using for both the same formalism. An STD is a special case of a Finite State Machine (FSM) that allows transitions between states either when an external or an internal event occurs to the system (according to his work, transitions in FSMs can only contain external events).

König defined a protocol as a structured exchange of messages. Then, he compared three approaches to modeling conversation policies, i.e. those based on STDs, FSMs and Petri nets. He observed that all approaches modeling conversations from the viewpoint of an observer are using either STD or petri nets, in contrast to those using FSM (or statecharts) that are representing the conversation from the viewpoint of a participating agent. For modeling a conversation from the point of view of a participating agent who receives and sends messages, König argued that a model supporting input and output operations is more suitable. When a conversation should be modeled from an observer's view, it is sufficient to use a model which is able to express that a message has been transmitted from one agent to another, like a transition in a STD or in a petri net. He chose STD aiming to model both activities and protocols, allowing also for object-oriented development.

König made the assumption that only two agents are involved in a protocol, i.e. the primary (who initiates the interaction) and the secondary. Moreover, the messages exchange is always synchronous, when one of them sends a message the other one is in a state of receiving a message (they cannot both be sending at the same time). Then, he defines an FSM for the observer and from it he derives the FSMs of the participants. In a next level (higher level of abstraction) he defines communication acts that can make use of the protocols in the form of STDs. Finally, in a third level he defines the activities of the agents that can invoke one or more communication acts and assume a wait state until the acts finish. The acts themselves can choose to execute one or more protocols and enter a wait state until they are finished. All these can only happen sequentially.

One of the most influential methodologies for AOSE also appeared at that time. The Gaia methodology [46,47] emphasized the need for new abstractions in order to model agent-based systems and supported both the levels of the individual agent structure and the agent society in the multi-agent (MAS) development process. Gaia added the notion of situatedness to the agent concept. According to this notion, the agents perform their actions while situated in a particular environment. The latter can be a computational environment (e.g. a website) or a physical one (a room) and the agent can sense and act in the environment.

MAS, according to Gaia, are viewed as being composed of a number of autonomous interactive agents that live in an organized society in which each agent plays one or more specific roles. Gaia defined the structure of a MAS in terms of a role model. The model identifies the roles that agents have to play within the MAS and the interaction protocols between the different roles.

The objective of the Gaia analysis phase is the identification of the roles and the modelling of interactions between the roles found. Roles consist of four attributes: responsibilities, permissions, activities and protocols. Responsibilities are the key attribute related to a role since they determine the functionality. Responsibilities are of two types: liveness properties – the role has to add something good to the system, and safety properties – the role must prevent something bad from happening to the system. Liveness describes the tasks that an agent must fulfil given certain environmental conditions and safety ensures that an acceptable state of affairs is maintained during the execution cycle. In order to realize responsibilities, a role has a set of permissions. Permissions represent what the role is allowed to do and, in particular, which information resources it is allowed to access. The activities are tasks that an agent performs without interacting with other agents. Finally, protocols are the specific patterns of interaction, e.g. a seller role can support different auction protocols. Gaia defined operators and templates for representing roles and their attributes and schemas for the abstract representation of interactions between the various roles in a system.

The Gaia2JADE process appeared in 2003 [28,29] and was concerned with the way to implement a multi-agent system with the emerging JADE framework using the Gaia methodology for analysis and design purposes. This process used the Gaia models and provided a roadmap for transforming Gaia liveness formulas to Finite State Machine diagrams. The JADE framework provided an object-oriented solution to building MAS and it became the most celebrated framework for building real-world software agents applications [3].

In 2004 there was also a proposal for the use of distilled statecharts (DSCs) for modeling mobile agents [16]. The proposal came along an object oriented implementation based on UML modeling. DSCs define some limitations to the language of Statecharts, e.g. only the OR-state decomposition is used, states do not have properties such as activities, therefore activities are only carried out under the form of atomic actions attached to transitions. If their source is not *start* and *history* states, transitions always include an *event*. In a later work, Fortino et al. [14] proposed a JADE implementation for DSC.

An important work on statecharts based agent development was that of Murray [31]. The latter, working for defining Robocup soccer player agents, explained that statecharts is a natural formalism for expressing multi-agent plans, as a player usually assumes a role, e.g. defender, attacker, goalkeeper, and the role's plan can be modeled as a sequence of states an agent passes through. A passing of a state can be the result of an (external) event or the completion of an activity, e.g. passing the ball. Moreover, as team players work together towards a common goal, they need to synchronize their actions and this can be modelled with one agent waiting for another agent to finish an action. Murray proposed a methodology and tool (StatEdit) for capturing this behavior based on a three layered approach:

– In the top level the different roles (*modes*) that the player can assume when active are represented as states and the transitions indicate a change of role
– In a middle level an agent chooses among a set of plans adding detail at each mode of the previous level. The states here capture the agent general activity and show where the player synchronizes its actions with other roles (e.g. wait for the center player to pass the ball and then shoot to score).
– On a bottom level of the hierarchy each activity of the role is detailed to specifc actions (e.g. acquire the ball and then kick towards the goal)

A similar layered approach was used later [22] for modeling the behavior of non-player characters in computer games. Murray also proposed an extension to statecharts with *synch* states for synchronizing the actions of different agents. His work, along with the previous one of Obst [33] both supported semi-automatic code generation for Robolog, a robot programming language based on Prolog.

Later, ASEME [40,43], uniquely among AOSE methodologies, used the statecharts formalism both for inter- and intra-agent control modeling. Moreover, it extended the statechart formalism by adding state-dependent variables. Thus, each state is associated with variables that it can monitor and change/update. To propose this extension, the authors were motivated by the Gaia methodology and the role's access to data structures with the read or write/update permissions [46]. Thus, ASEME proposed the addition of the Var property to the statechart nodes. The different states can be connected with variables that can be used for exchanging information.

Definition 3. *The tuple (L, δ) defined in Definition 2 is extended by adding Var to L:*

– $L = (S, \lambda, Name, Activity, Var)$ *is an ordered rooted tree structure representing the states of the statechart, where:*
  - *Var is a mapping from nodes to sets of variables.* $var(l)$ *stands for the subset of local variables of a particular node* l.

According to ASEME, a state name that starts with the string "send" implies an inter-agent message sending behavior for the state's activity. A send state has only one exiting transition and its event describes the message(s) sent. Similarly, a state name that starts with the string “receive” implies that the activity of the state should wait for the receipt of one or more inter-agent messages. The type and quantity of the expected messages can be implied by the events monitored by the transition expressions that have this state as source. The events that can be used in the transition expressions can be:

– a sent or received (or perceived, in general) inter-agent message,
– a change in one of the executing state’s variables (also referred to as an intra-agent message),
– a timeout, and,
– the ending of the executing state activity (empty event).

This formalism allows also for environment-based communication by defining state activities that monitor for a specific effect in the environment. This effect can be expected to be caused by any other agent or a particular agent. Such activities can be, for example, “wait for someone to appear” or “wait until my counterpart lifts the object” respectively.

ASEME defines protocols as statecharts where the participating roles are defined as orthogonal components. See Fig. 3 as an example. Two roles are connected to this protocol, the service requester (sr) and the service provider (sp). The reader will notice these two roles as orthogonal components in the *RequestForServices* protocol state. The requester sends a request *message* using the *Request* performative whose variables are the sending and receiving agents (we use the abbreviation sr for service requester and sp for service provider) and the *request*, which can be an object for object oriented implementations or a query for logic-based implementations. On the other hand, the service provider waits to receive this message, then processes the request and either replies with a *Refuse* (the service is refused for this agent), *Failure* (failed to reply), or *Inform* (with the results of the computation) performative. Note that the protocol terminates for both roles after a timeout of 10000 ms. A similar model also appears in the work of Seo et al. [39] for buying products.

The work of Moore [27] supported the possibility of an agent getting involved in a *sub-dialog* when in a dialog. In ASEME, the model for describing such dialogs is the inter-agent control model. Moore supposed that the agent has access to a repository of dialogs and dynamically selects a sub-dialog model whenever an incoming message is not permitted by the existing dialog but is permitted by another in the repository. In the intra-agent control model, ASEME allows for this possibility as all roles the agent can participate in can be instantiated as orthogonal components. Information between orthogonal components can be exchanged through the use of common variables and their usage in transition expressions, thus, a given protocol can remain in a given state until some information becomes available (an implicit intra-agent message).

Another feature of ASEME is the catering for *embedded dialogs* in an agent’s design, i.e. in its intra-agent control model. Dialogs occur when an agent participates in an agent interaction protocol. Instances of dialogs contained entirely within other dialogs are said to be embedded [25]. ASEME defines that when a

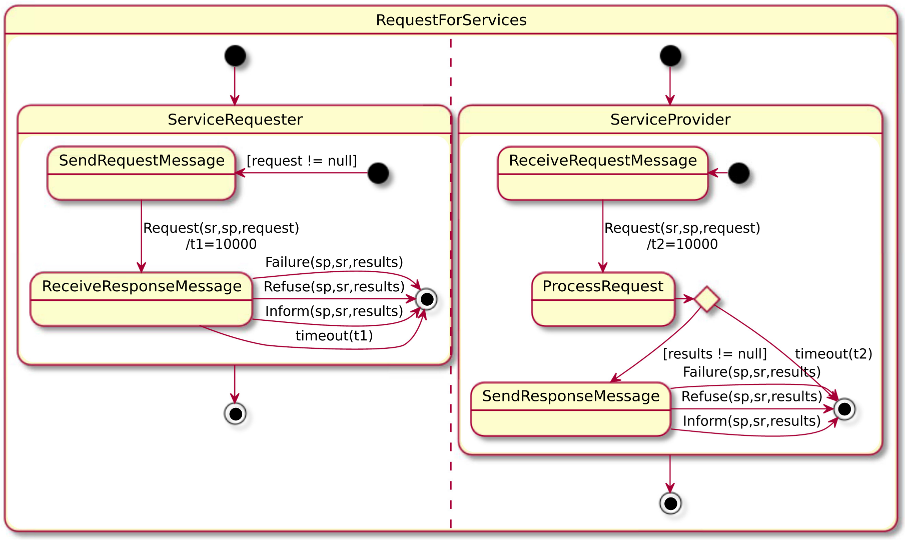

Fig. 3. Statechart representation of a protocol for requesting a service. The diamond shape represents a condition state.

role in a protocol model is integrated in an intra-agent control model, the protocol role OR-state is inserted as-is in the intra-agent control. Then, the designer is free to define the activities of the basic states. The designer can even select to expand a basic state and turn it to an OR-state.

Thus, the reader can see a broker agent in Fig. 4 realizing the protocol defined in Fig. 3. The broker agent realizes the service provider role of the service request protocol. However, for defining the process request state activity, the designer decided that the broker will initially perform a service matching activity and then either invoke a web service, or employ an embedded dialog, i.e. the service request protocol, this time as a service requester. Note that the transition expressions have been omitted in Fig. 4 so as not to clutter the diagram.

Recently, researchers explored the translation of agent models defined using the Distilled StateCharts (DSC) [13,16] into a Belief-Desires-Intentions (BDI) framework [15], including a BDI-like code generation feature. BDI is an example of an agent architecture including an execution paradigm besides ontological features [36]. BDI advocates the fact that an agent first senses its environment and updates its *beliefs*, then it searches possible *desires*, i.e. goals that are valid in this environment state, and, finally, selects some of these desires to actively pursue. The latter are now its *intentions*.

Statecharts can be used for modelling the dynamic behaviour of a BDI agent. See for example the execution model followed by 3APL [8], a BDI-based agent development language, modelled as a statechart in Fig. 5. The lifecycle of this agent starts in the *ReceiveMessage* state. Then, as soon as a message arrives, or another monitored for event occurs, the BDI agent enters the

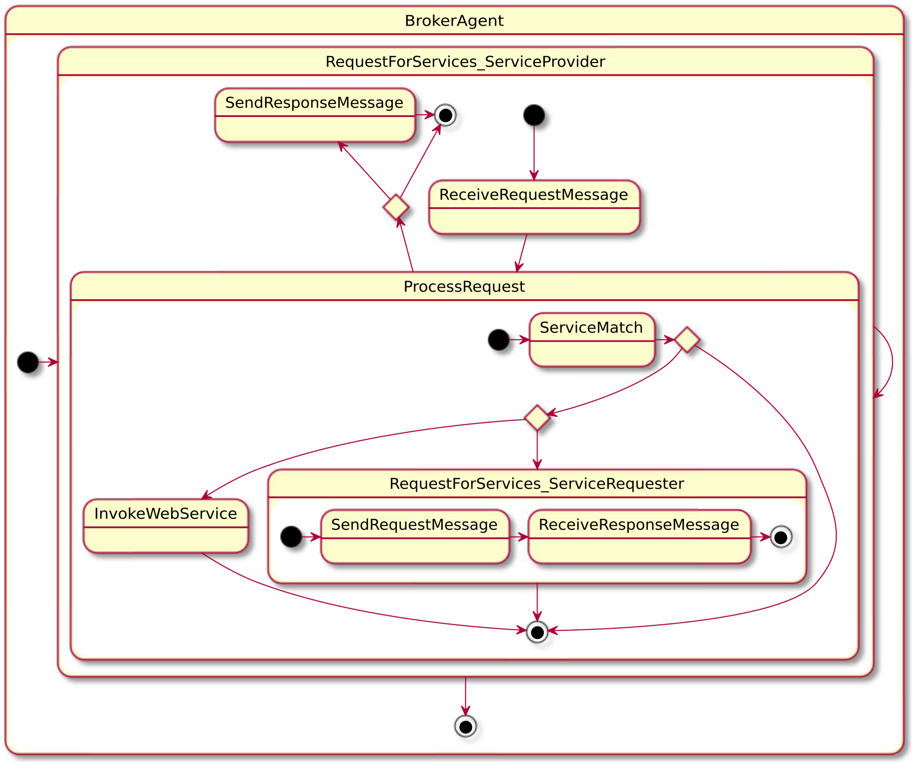

Fig. 4. Statechart representation of a Broker agent embedding a dialog in another dialog.

ApplyGoalPlanningRules OR state. Within that OR state, more specific activities match the goals with rules, select rules matching the agent's beliefs and apply a goal planning rule. The next OR state, i.e. *ApplyPlanRevisionRules*, and its substates find rules matching to the plans, select rules matching the agent's beliefs and apply the selected plan revision rule. Finally, the agent reaches the *ExecutePlan* state that, depending on the selected plan, may send a message, take an external action or an internal (or mental) action, or do nothing. After finishing the plan execution the agent returns to his message receiving state. This is an example of how someone can use statecharts to coordinate the agent's capabilities and to accommodate a well-known type of architecture in a platform independent manner, i.e. the way to implement this model is not yet chosen at this time.

In another work, researchers provided the Kouretes Statechart Editor (KSE) CASE tool for authoring robotic behaviours [44]. Given existing bottom level functionalities [31], e.g. kick the ball, the modeler could define a robotic behaviour visually and immediately generate the code and upload it to a humanoid (Nao) robot.

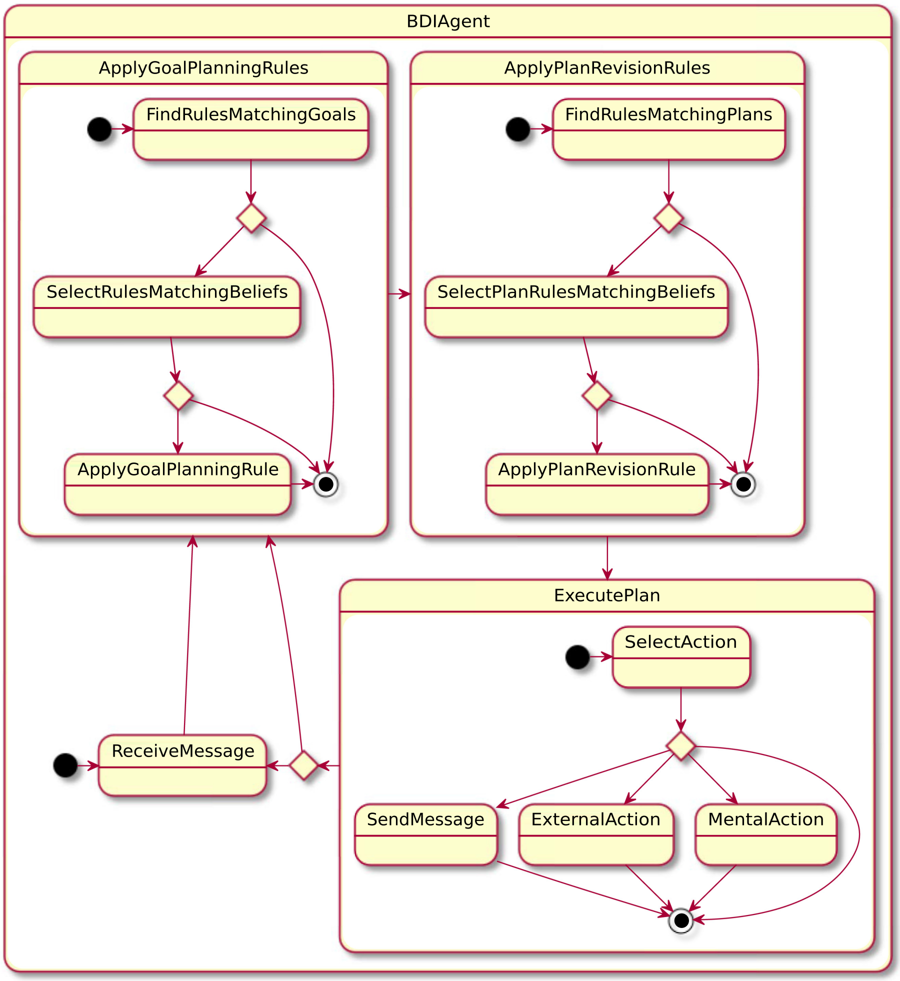

Fig. 5. Statechart representation of a BDI agent [41].

4 Discussion

The statecharts main added value is the capability of the language to capture both the static (activities and variables) and dynamic aspect of a system [19,20]. Thus, one can have a unique design model and use it to generate code for diverse platforms.

AOSE researchers have argued on other pros and cons of statecharts. Mainly inspired by the work of Paurobally et al. [35] we present some of their advantages (+) and disadvantages (-):

+ States and processes can be treated equally allowing an agent to refer and reason about the state of an interaction
+ Statechart notation is more amendable for extension thanks to their simple semantics
+ Visual models are easier to conceive and display [16]
+ Engineers familiar with UML can start working with them immediately [37]
- Participating roles are not shown explicitly
- Compound transitions are not shown in detail
- There is a question of completeness

One of our findings by working with statecharts is that agent behavior specification is not a trivial task. The development of the simplest possible player in Robocup took a statechart with 99 states in a hierarchy with a depth of 17 [34,44]. This demonstrated the added value of the ASEME methodology as it allows for the automatic transformation of Gaia liveness formulas to a statechart [42], which is at least a "good start", as opposed to starting the design directly with a statechart CASE tool, as was the case of StatEdit [31], or using a flat statechart model with no hierarchy, such as the plan diagrams of MaSE [10].

Proposing radical extensions to the language of statecharts may seem to facilitate or enable new features, e.g. as in the case of ANML, however, it renders them incompatible with existing CASE tools and they may become difficult for mainstream software engineers to learn and use [37].

Some times, and especially in works that do not adopt the orthogonal components of statecharts (i.e. AND-states), it is not obvious how one develops an agent realising more than one protocols simultaneously, and/or how to combine them with other agent capabilities.

The ASEME inter and intra-agent control models, being derived by Gaia formulas, do not use the possibility of the state transitions to traverse levels or the history connectors. If the developers, however, choose to introduce these features to the statechart they lose the connection of the design phase models (the statecharts) to the analysis phase models (i.e. the role model and the Gaia formulas). This situation can impact the tracing of software features to their requirements and has been reported as the "round-trip" problem [38]. The acquired experience after modeling a number of systems for software and robotic agents shows that the choice to not use state transitions traversing levels or the history connectors does not hinder the possibility to model complex systems, on the other hand, important engineering concepts, such as comprehension, modularity and reusability, are enhanced. The same has been reported by the more recent work on Armax statecharts for modeling robotic behaviour [45].

5 Future Directions

The future holds many challenges. Regarding the use of statecharts, agents and autonomous systems continuously face the possibility of an unexpected (at design time) event to happen while they are in operation. Unexpected means that either a known event happens that the system is unable to handle at its current state (unexpected at that time), although it is related to its operation, or an unknown event happens (totally unexpected), see Marron et al. [24] for a detailed definition. Although there are some hacks for ad-hoc catering for this issue, such as having a default handler for incoming messages not handled by a defined behaviour that replies with information about the services offered by the agent, this is a valid research direction.

In the area of design for autonomy (empowerment, self-management and self-regulation) it is very interesting to research how an inter- or intra-agent control can self-evolve over time. Evolution may be triggered through introspection or through the desire to maximize or fine-tune an agent's performance. For example, a robot may have a failing limb, it may need to fine tune its grasp to manage its best with the available functionality. Another example is related to the previously presented broker agent. What happens if, while usually its service matching activity successfully matches 99% of the requests, suddenly, and for a significant period, it matches only 30% of the requests. Now the agent needs a strategy to mitigate, e.g. to reboot its system, or update its services repository. Another kind of evolution is to evolve the statechart itself. Researchers are already delving into this area with results only for flat statecharts until now [18].

Recently, researchers proposed the concept of the property statecharts [26] for expressing and enforcing safety criteria in statecharts. Safety properties have been defined in AOSE, and the Gaia methodology's role model [46], however, statechart-based design models have not yet fully realised this feature, especially those leading to object-oriented implementations. Property statecharts are monitoring the events generated by the execution of normal statecharts and safeguard conditions. For example, and in the area of *smart contracts*, Mens et al. [26] have given an example, where an agent A signs a Service Level Agreement (SLA) with agent B. The SLA dictates that whenever A receives a request from B, then A must reply within 1 h. The property statechart gets in the monitoring state whenever A receives a request from B. If the A's state for sending a reply to B is not exited within 1 h the contract is considered violated. It would be very interesting to adapt this idea to safety properties of agents.

To realize implementations of agents in the modern open systems [21], agents need to use predefined protocols to interact. However, when diverse stakeholders come in, they need to work the protocols with their own algorithms and/or goals. Currently, protocols focus on defining sequences of exchanged messages. Adopting the point of view of the ASEME methodology [40,43], where protocols are regarded not as simple communication protocols that determine how data is transmitted (as in telecommunications and computer networking), but as their higher level abstractions used by humans, where protocols define *codes of behaviour* (or *procedural methods*), we can use statecharts for defining them. Thus, a protocol does not only answer the question of what messages are allowed but also what actions the participants need to do within the protocol. In this context, an important direction is towards defining new design patterns, that on the one hand will allow the developers to re-use existing protocol parts and logic defined in the open system; and on the other hand to customize key functionality or capabilities according to their needs and/or goals.

Thus, when defining open systems, or even proprietary systems, the use of statechart repositories would lead to the simplification of the statechart-based agent development. Consider for example, the Robocup Player agent that we referred to in the discussion above. It would be much easier to develop this agent if some parts of its statechart or intra-agent control model were reused from local or public repositories.

For example, 28 students taking the Autonomous Agents class at an Electrical and Computer Engineering school of the Technical University of Crete were asked to develop a robocup player in one of the 2-h laboratory sessions of the class. The students worked in small teams of two or three people per team. The students first went through a quick tutorial on using the ASEME CASE tool, which demonstrated the development of a Goalie behavior for the Nao robot. This included the Gaia formulas for the goalie role, and its IAC model. Then, they were asked to use the existing functionalities of the Goalie (scan for the ball, kick the ball, approach the ball, etc) to develop an *attacker* behavior using KSE. Thus, the students did not have to develop the robot functionalities. They defined the attacker role's liveness and then edited the generated statechart, i.e. they defined variables and transition expressions. All student teams were able to deliver the requested Attacker behavior and enjoyed watching their players in a game (for more information the reader can consult [43,44]).

A step forward would be to have the developers not reuse just activities of states (as they did in the above experiment) but whole statechart components (including transition expressions) as modules. Modules have also been referred to as capabilities in the AOSE community [5,43]. Modular programming has been identified as the ultimate aim of agent programming languages and developing frameworks be they declarative or imperative [7].

6 Conclusion

Statecharts and agent technology are quite close, as the reader may have already found out. The future holds more prospects for both areas but also for their cooperation. Statechart-based agent modeling has been used for developing software and physical agents, object-oriented or logic-based agents, agents communicating through message exchanging or through a blackboard.

We presented several future directions, mainly for the intra-agent's control, on one hand monitoring events that are essential for the agent's successful operation and detecting failing capabilities, and on the other hand safeguarding restrictions and contracts. Statecharts are a still evolving paradigm [6,24,26,45] and modern AOSE works use it [15,43].

In the future, this work can be expanded by adding a survey on the real-world multi-agent systems that have been developed using statechart-based designs. This is interesting as it will flesh out the relevant application domains and gather the experience that goes along with developing real-world systems. Moreover, more related work from the statechart research community can be added, as the present work was concerned with the works from the autonomous agents and MAS community.

References

1. Austin, J.L.: How To Do Things With Words. Harvard University Press, Cambridge (1975)
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley Series in Agent Technology. Wiley, Hoboken (2007)
3. Bordini, R.H., et al.: A survey of programming languages and platforms for multi-agent systems. Informatica **30**(1) (2006)
4. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM (JACM) **30**(2), 323–342 (1983)
5. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible BDI agent modularization. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) ProMAS 2005. LNCS (LNAI), vol. 3862, pp. 139–155. Springer, Heidelberg (2006). https://doi.org/10.1007/11678823_9
6. Broad, A., Argall, B.: Path planning under interface-based constraints for assistive robotics. In: Proceedings of the 26th International Conference on Automated Planning and Scheduling, ICAPS 2016, pp. 450–458. AAAI Press (2016)
7. Dastani, M.: Programming multi-agent systems. Knowl. Eng. Rev. **30**(4), 394–418 (2015). https://doi.org/10.1017/S0269888915000077
8. Dastani, M., van Birna Riemsdijk, M., Meyer, J.-J.C.: Programming multi-agent systems in 3APL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming. MSASSO, vol. 15, pp. 39–67. Springer, Boston, MA (2005). https://doi.org/10.1007/0-387-26350-0_2
9. DeLoach, S.A., Garcia-Ojeda, J.C.: O-MaSE: a customisable approach to designing and building complex, adaptive multi-agent systems. Int. J. Agent-Oriented Softw. Eng. **4**(3), 244–280 (2010)
10. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. Int. J. Softw. Eng. Knowl. Eng. **11**(03), 231–258 (2001)
11. Dunn-Davies, H., Cunningham, R., Paurobally, S.: Propositional statecharts for agent interaction protocols. Electron. Notes Theor. Comput. Sci. **134**, 55–75 (2005)
12. Eilenberg, S.: Automata, Languages, and Machines. Academic Press (1974)
13. Fortino, G., Garro, A., Mascillaro, S., Russo, W.: Using event-driven lightweight DSC-based agents for MAS modelling. Int. J. Agent-Oriented Softw. Eng. **4**(2), 113–140 (2010)
14. Fortino, G., Rango, F., Russo, W.: Statecharts-based JADE agents and tools for engineering multi-agent systems. In: Setchi, R., Jordanov, I., Howlett, R.J., Jain, L.C. (eds.) KES 2010. LNCS (LNAI), vol. 6276, pp. 240–250. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15387-7_28
15. Fortino, G., Rango, F., Russo, W., Santoro, C.: Translation of statechart agents into a BDI framework for MAS engineering. Eng. Appl. Artif. Intell. **41**, 287–297 (2015)
16. Fortino, G., Russo, W., Zimeo, E.: A statecharts-based software development process for mobile agents. Inf. Softw. Technol. **46**(13), 907–921 (2004)
17. Girault, A., Lee, B., Lee, E.A.: Hierarchical finite state machines with multiple concurrency models. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **18**(6), 742–760 (1999)

18. Goldsby, H.J., Cheng, B.H., McKinley, P.K., Knoester, D.B., Ofria, C.A.: Digital evolution of behavioral models for autonomic systems. In: Proceedings of the 5th IEEE International Conference on Autonomic Computing (ICAC 2008), pp. 87–96. IEEE Computer Society, Los Alamitos (2008)
19. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
20. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. (TOSEM) **5**(4), 293–333 (1996)
21. Huynh, T.D., Jennings, N.R., Shadbolt, N.R.: An integrated trust and reputation model for open multi-agent systems. Auton. Agents Multi-Agent Syst. **13**(2), 119–154 (2006)
22. Kienzle, J., Denault, A., Vangheluwe, H.: Model-based design of computer-controlled game character behavior. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 650–665. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_44
23. König, R.: State-based modeling method for multiagent conversation protocols and decision activities. In: Carbonell, J.G., Siekmann, J., Kowalczyk, R., Müller, J.P., Tianfield, H., Unland, R. (eds.) NODe 2002. LNCS (LNAI), vol. 2592, pp. 151–166. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36559-1_13
24. Marron, A., Limonad, L., Pollack, S., Harel, D.: Expecting the unexpected: developing autonomous-system design principles for reacting to unpredicted events and conditions (2020)
25. McBurney, P., Parsons, S.: Dialogue games for agent argumentation. In: Simari, G., Rahwan, I. (eds.) Argumentation in Artificial Intelligence, pp. 261–280. Springer, US, Boston (2009). https://doi.org/10.1007/978-0-387-98197-0_13
26. Mens, T., Decan, A., Spanoudakis, N.I.: A method for testing and validating executable statechart models. Softw. Syst. Model. **18**(2), 837–863 (2018). https://doi.org/10.1007/s10270-018-0676-3
27. Moore, S.A.: On conversation policies and the need for exceptions. In: Dignum, F., Greaves, M. (eds.) Issues in Agent Communication. LNCS (LNAI), vol. 1916, pp. 144–159. Springer, Heidelberg (2000). https://doi.org/10.1007/10722777_10
28. Moraïtis, P., Petraki, E., Spanoudakis, N.I.: Engineering JADE agents with the Gaia methodology. In: Carbonell, J.G., Siekmann, J., Kowalczyk, R., Müller, J.P., Tianfield, H., Unland, R. (eds.) NODe 2002. LNCS (LNAI), vol. 2592, pp. 77–91. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36559-1_8
29. Moraitis, P., Spanoudakis, N.: The GAIA2JADE process for multi-agent systems development. Appl. Artif. Intell. **20**(2–4), 251–273 (2006). https://doi.org/10.1080/08839510500484249
30. Moulin, B., Chaib-Draa, B.: An overview of distributed artificial intelligence. In: O'Hare, G.M., Jennings, N.R. (eds.) Foundations of Distributed Artificial Intelligence. Wiley (1996)
31. Murray, J.: Specifying agent behaviors with UML statecharts and StatEdit. In: Polani, D., Browning, B., Bonarini, A., Yoshida, K. (eds.) RoboCup 2003. LNCS (LNAI), vol. 3020, pp. 145–156. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25940-4_13
32. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: Zeus: a toolkit for building distributed multiagent systems. Appl. Artif. Intell. **13**(1–2), 129–185 (1999)
33. Obst, O.: Specifying rational agents with statecharts and utility functions. In: Birk, A., Coradeschi, S., Tadokoro, S. (eds.) RoboCup 2001. LNCS (LNAI), vol. 2377, pp. 173–182. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45603-1_18

34. Paraschos, A., Spanoudakis, N.I., Lagoudakis, M.G.: Model-driven behavior specification for robotic teams. In: Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 171–178. International Foundation for Autonomous Agents and Multiagent Systems (2012)
35. Paurobally, S., Cunningham, J., Jennings, N.R.: Developing agent interaction protocols using graphical and logical methodologies. In: Dastani, M.M., Dix, J., El Fallah-Seghrouchni, A. (eds.) ProMAS 2003. LNCS (LNAI), vol. 3067, pp. 149–168. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25936-7_8
36. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: KR, pp. 473–484 (1991)
37. Riemenschneider, C.K., Hardgrave, B.C., Davis, F.D.: Explaining software developer acceptance of methodologies: a comparison of five theoretical models. IEEE Trans. Softw. Eng. **28**(12), 1135–1145 (2002)
38. Selic, B.: The pragmatics of model-driven development. IEEE Softw. **20**(5), 19–25 (2003)
39. Seo, H.-S., Araragi, T., Kwon, Y.R.: Modeling and testing agent systems based on statecharts. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 308–321. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30233-9_23
40. Spanoudakis, N., Moraitis, P.: An agent modeling language implementing protocols through capabilities. In: Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, vol. 02, pp. 578–582. IEEE Computer Society (2008)
41. Spanoudakis, N.: The agent systems engineering methodology (ASEME). Ph.D. thesis, Paris Descartes University (2009)
42. Spanoudakis, N., Moraitis, P.: Gaia agents implementation through models transformation. In: Yang, J.-J., Yokoo, M., Ito, T., Jin, Z., Scerri, P. (eds.) PRIMA 2009. LNCS (LNAI), vol. 5925, pp. 127–142. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-11161-7_9
43. Spanoudakis, N.I., Moraitis, P.: The ASEME methodology. Int. J. Agent-Oriented Softw. Eng. (in press)
44. Topalidou-Kyniazopoulou, A., Spanoudakis, N.I., Lagoudakis, M.G.: A CASE tool for robot behavior development. In: Chen, X., Stone, P., Sucar, L.E., van der Zant, T. (eds.) RoboCup 2012. LNCS (LNAI), vol. 7500, pp. 225–236. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39250-4_21
45. Wächter, M., Ottenhaus, S., Kröhnert, M., Vahrenkamp, N., Asfour, T.: The ArmarX statechart concept: graphical programing of robot behavior. Front. Robot. AI **3**, 33 (2016)
46. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Auton. Agents Multi-Agent Syst. **3**(3), 285–312 (2000)
47. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: the Gaia methodology. ACM Trans. Softw. Eng. Methodol. **12**(3), 317–370 (2003). https://doi.org/10.1145/958961.958963