Protocol Design Patterns for Statecharts-Based Open MAS Development

Nikolaos Spanoudakis[1][0000−0002−4957−9194], Charilaos Akasiadis[2][0000−0003−0785−4036], Georgios Kechagias[1], and Georgios Chalkiadakis[1][0000−0002−0716−2972]

[1] Technical University of Crete, Chania, Greece
{nispanoudakis,gkechagias,gchalkiadakis}@tuc.gr
[2] Institute of Informatics and Telecommunications, NCSR 'Demokritos'
Patr. Gregoriou E & 27 Neapoleos Str, Agia Paraskevi, 15341 Greece
cakasiadis@iit.demokritos.gr

Abstract. In this paper we define two novel design patterns that can be used with statecharts-based agent modeling in many real-world domains that call for open MAS architectures. The first pattern is based on the well-known factory design pattern, and the second on the composite pattern. The applicability of our proposed patterns is demonstrated with ASEME, an agent-oriented software engineering methodology that uses statecharts for the inter- and intra-agent control models. The latter can also pattern up well with the FIPA standards-compliant JADE agent platform. We showcase the patterns usage through an application for the modern vehicle-to-grid (V2G) and grid-to-vehicle (G2V) energy transfer problem domain.

Keywords: design patterns · statecharts · engineering multiagent systems · agent oriented software engineering.

1 Introduction

Modern Multi-Agent Systems (MAS) applications areas, such as the emerging Smart Grid [3, 9, 36] pose challenges to the Agent-Oriented Software Engineering (AOSE) community. In the Smart Grid, buildings, but also vehicles, become active consumers and producers of energy, and need to be integrated into this emerging energy and information network. Not only is the Smart Grid a marketplace with diverse consumers and producers, it is also a dynamic setting where heterogeneous devices appear and need to connect [25, 44]. To date, several Smart Grid-related business models and information systems' architectures have been proposed, but they do not adhere to particular standards [17].

Such energy markets naturally reflect systems where a single player cannot force others to use her products; players or stakeholders can come along their own business models; and stakeholders can have diverse goals in negotiating their consumption and offer. Moreover, these systems allow for pro-activeness on the part of the players who pursue their goals; and sociability, as entities can form

dynamic partnerships or coalitions, but also react and/or adapt to a changing dynamic environment. In addition, it is natural for participants to be generally able to freely join and leave the system at any time. All these characteristics point to *open multiagent systems* and agent technology in general [22, 47].

Now, to realize an open system, 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. This has been the focus of the well-known AUML Agent Interaction Protocol model [31] that has been adopted by many AOSE methodologies, i.e. Tropos [8], PASSI [12], ADELFE [34], Mas-CommonKADS [23] and MESSAGE [20]; however, developers still need to define the appropriate behaviours for their agents from scratch.

Early works [24, 37] in the AOSE domain focused on defining patterns of usage of objects for addressing agent-specific concepts, including protocols. Kendall et al. [24] identify the need for collaboration patterns via the use of coordination protocols, thus potentially allowing the agent to determine its behaviour based on the state of the employed protocol, and the possibility to be engaged in several conversations simultaneously. These early directives have not, unfortunately, materialized in modern AOSE methodologies to date.

In our work in this paper, we adopt the point of view of the ASEME methodology [45], 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*). Thus, a protocol does not only answer the question of what messages are allowed but also what actions do the participants need to do within the protocol. Then, we define two new design patterns, that on the one hand allow the developers to re-use the protocol parts and logic defined in the framework; and on the other hand to customize key functionality or capabilities according to their needs and/or goals.

We demonstrate the use of these design patterns through the system that we developed for the Smart Grid focusing in the Vehicle-to-Grid (V2G)/ Grid-to-Vehicle (G2V) problem, where electric vehicles charge using the grid (G2V) but can also give back power to the grid when they are charged and prices soar or there is a shortage.

The rest of this paper is structured as follows: In Section 2 we present the necessary background and discuss related work. In Section 3 we give the problem statement and then in Section 4 we define the novel design patterns that we propose. In Section 5 we discuss our findings and conclude.

2 Background and Related Work

To demonstrate the added value of this work we will use JADE [5], a standardized, state of the art, platform that has proved its worth in real world applications in the smart grid [38, 4, 1, 42, 26, 35, 18]. In JADE, the developer defines state-

machine-like control of the agent's behaviours, and respects the standards set by FIPA[3].

JADE naturally fits with development methodologies whose design phase models are statecharts. Statecharts have been used by a number of AOSE methods or methodologies [43, 45, 29, 33, 15, 28]. A recent one is ASEME [45], which also allows to automatically transform the design phase models to JADE code [41]. We present an overview of JADE and ASEME in some detail below.

2.1 The Java Agent Development Framework

JADE [5] is a software development framework for distributed MASs. It provides a rich Java API, and a collection of features that enable peer to peer asynchronous agent communication, agent service discovery, management of ontologies and context languages, and debugging and monitoring tools. In JADE, the activities an agent needs to perform are typically carried out within *behaviours*. Each behaviour performs its designated operation by executing the method `action()`, inherited from the `Behaviour` class.

Behaviour is abstractly defined as the pattern of interaction of an entity (agent) with its environment [48]. In JADE, a behaviour refers to a minimum chunk of activity that the agent can undertake. JADE provides a behaviour library that enables the composition of more complex actions by combining different behaviours in various ways. For instance, the *FSMBehaviour* is a composite behaviour that executes its sub-behaviours according to a user-defined Finate State Machine (FSM) with the sub-behaviours as states.

Each agent has a built-in list that contains the active behaviours that are scheduled for execution according to the order of their insertion. New behaviours can be added into the behaviour list. Through a non-preemptive behaviour scheduling, a behaviour is scheduled for execution, and when it is activated, its `action()` method is invoked. It runs until finishing. Then the behaviour's `done()` method is checked and if it returns `true` the behaviour is removed from the scheduler, else it will re-execute in a next iteration.

In terms of data exchange, sharing and storage, JADE behaviours can use the *DataStore* structure. This makes the developed behaviours completely independent from a specific application, i.e. they are re-usable.

Agents communicate using messages. *Performative* verbs are used to utter the specific intention that a message carries [48]. Communicative acts such as *request*, *inform*, *propose* are assigned to the *Performative* parameter of a FIPA-ACL message. FIPA does not mandate the use of a specific content language for messages, but, through an independent specification, proposes the FIPA-SL language, a human-readable string-encoded first-order modal language which can represent propositions, objects, and actions.

Importantly, in agent communication there typically exist *predefined message sequences* that can be applied in several situations that share the same communi-

[3] FIPA, the Foundation for Intelligent Physical Agents (http://www.fipa.org), is an IEEE Computer Society standards organization for software agent technology.

cation pattern regardless of the application domain [5]. Such message sequences are referred to as *protocols*.

2.2 ASEME

The Agent Systems Engineering Methodology (ASEME [45]) is a methodology for developing multi-agent systems. It builds on existing languages, such as Unified Modeling Language (UML) and statecharts [21], to represent system analysis and design models. It is agent architecture and mental model independent, thus the designer can select the architecture type and the mental attributes of the agent that they prefer (allowing also for heterogeneous agent architectures). Moreover, ASEME supports a modular agent design approach and uses the intra-agent and inter-agent control concepts. The first defines the agent's behavior by coordinating the different modules that implement its capabilities, while the latter defines the protocols that govern the coordination of the agent society.

ASEME uses two quite common abstractions for modeling agents: *capability* and *functionality*. Busetta et al. [10] view capability as "a cluster of plans, beliefs, events and scoping rules over them". Braubach et al. [7], extended this idea and proposed that capabilities can contain sub-capabilities and have at most one parent capability. They define the agent concept as an extension of the capability concept aggregating capabilities. In the Prometheus methodology [32], each functionality identified in the analysis phase ends up mapping to a capability in the design phase. Capability in the Agent Modeling Language (AML) [46] is a concept used to model an abstraction of a behaviour in terms of its inputs, outputs, pre-conditions, and post-conditions. A behaviour is the software component, and its capabilities are the signatures of the methods that the behaviour realizes, accompanied by method pre-conditions and post-conditions. This approach is similar to service oriented architectures, and thus considers the agent as an aggregation of services.

In ASEME, the agent coordinates its capabilities in the intra-agent control model to define its *behaviour*. Capabilities are decomposed to simple *activities*. For example, a decision making capability fulfills the task of finding where to recharge this evening. The problem is decomposed to activities, e.g. of finding which stations are nearby, what are their available charging slots, rank them, etc. Each activity is realized by a specific *functionality*, i.e. a specific technique like an algorithm, invocation of a service, etc. The same capability can be implemented with different functionality types: an agent can have a *decision making task* based on an *argumentation-based functionality*, while another implementation of the same capability could be based on a different functionality, e.g. *multi-criteria-based functionality*.

The inter-agent control model defines the capability of an agent to participate as a role in a specific protocol. ASEME allows the seamless integration of the inter-agent control model in the intra-agent control as they follow the same formalism—i.e., statecharts [21]. The statecharts formalism does not have the limitations (limited scalability, explosion of states) posed on other formalisms

such as petri-nets [27]. Statecharts allow to model the activities that form a behaviour together with the events that allow state transitions in one or more of the participating roles. Therefore, in this work we use the statechart formalism to define our open protocols and design patterns. In a sense, the statechart is the adopted behavioral pattern by ASEME. Behavioral patterns are applied when an object's behavior depends on its state, which also changes at run-time [19].

2.3 Design Patterns for MAS

A design pattern for MAS is defined by *(a)* its name, *(b)* the problem and the context in which it appears, *(c)* the solution, which can be the objects, interfaces, and other information, and *(d)* the results and possible tradeoffs [24, 37].

Existing works [24, 37] focused in defining patterns of usage of objects for addressing the agent-specific concepts, including protocols. Kendall et al. [24] identified the need for collaboration patterns via the use of coordination protocols, where the agent would be able to determine its behaviour based on the state of the employed protocol, and the possibility to be engaged in several conversations simultaneously.

Existing AOSE methodologies focused in the communication part of the protocols leading to the first patterns that were applied to methodologies (e.g. PASSI [13]) and accommodated the use of specific protocol patterns that were identified by FIPA (e.g. *request*, *contract-net*, etc). Such patterns are used in the high methodological level and are implemented through existing support in platforms (e.g. JADE [6, 5]).

In our work, we go one step further to define specific patterns for *stateful* protocols based on their employment by an open MAS.

3 Problem Statement

To define our architecture, we assume that agents coexist in a micro-grid infrastructure that can be interconnected with other parts of the Grid through distribution and transmission networks. When the micro-grid requires additional power that can not be generated locally, it can import it, while, when it has a local energy surplus, it can export it to the Grid and create additional profits for its electricity producers, according to energy market regulations [25]. Figure 1 provides a part of the overall architecture, including the stakeholders directly related to the electric vehicles (EVs), i.e. EVs, charging stations and station recommenders. the latter may represent a regulatory service, a firm of charging stations, etc. The architecture also includes their interactions and their basic components.

The corresponding agent types system are: the *(a)* Electric Vehicle agents (EV), the *(b)* Charging Station agents (CS) and the Station Recommender agent (SR). Note that, each agent type consists of certain "private" sub-modules, whose specific functionality can further differentiate agent behaviours.

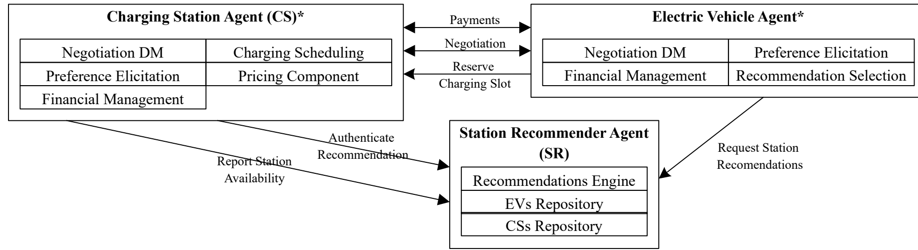

Fig. 1. Part of the architecture for the V2G/G2V problem. Asterisks (*) denote agent types with multiple instances.

EV agents aim to optimize a utility function set by the EV owner— e.g., always have enough energy to realize the next trip, achieve so by the minimum cost, etc. The *EV agent* monitors the driver's activities, models and predicts their future behavior and needs, contacts charging stations to schedule battery charging, seeks profit from participation in V2G activities, and, thus, engage in negotiations with the charging stations. Building blocks (corresponding to specific functionalities) are: a *Preference Elicitation* module, responsible for monitoring the habits and behaviour of the driver, and forecast future (charging) preferences; a *Financial Management* module, responsible for making EV charging payments, and receiving payments from participation in V2G activities; a *Negotiation Decision Making* module containing the algorithms and logic according to which the agent conducts negotiations; and a *Recommendations Selection* module, that contains agent-specific algorithmic components determining which charging recommendation to select. An *EV agent* communicates with the *SR agent* to receive recommendations and the *CS agents* to reserve station slots.

CS agents manage the physical gateways (i.e. connectors, parking slots) by which EVs connect to the grid and create profit by charging their batteries. They can also negotiate with *EV agents* regarding an existing charging agreement, to change some of the parameters so as be able to schedule the charging of additional vehicles. This leads to better utilization of the station infrastructure, and maximizes its profit. A *CS agent* contains a *Charging Scheduling* module, the algorithmic component responsible for schedule charging/discharging activities over a predefined planning horizon; a *Negotiation Decision Making* module for conducting negotiations; a *Pricing* module that calculates the EV payments; a *Preference Elicitation* module that monitors charging slots usage and updates the prices for each of them according to the needs of the station owner; and a *Financial Management* module that manages the station's finances and handles monetary payments. A *CS Agent* communicates with the *SR*, the *EV* agents and other grid-related agents.

The *SR agent* recommends to EVs a subset of the available CS and charging slots that match most with the EVs preferences (e.g. time of arrival/departure, distance, battery state). This agent can be also augmented to take into account various grid constraints in order to help avoid herding effects for example. It

consists of a *Recommendations Engine* module that makes charging stations recommendations, an *EV Repository* module storing information about the past EV behaviour in order to utilize it for future recommendations, and a *Charging Station Repository* of registered charging stations. It communicates with the *CS*, *EV*, and other agents offering grid-related services.

The architecture above poses several challenges for the rest of the development process. Clearly, there is a need to accommodate different methods for decision-making based on user preferences, or on the business model of a stakeholder, or on the agent that implements a protocol role. As an example, both *EV* and *CS* agents have the capability to negotiate, however, it is obvious that they will most likely employ different algorithms to do so.

Thus, we need to cater for agents following protocols to realize their goals in the system, while being able to define their own algorithms, policies or business rules. These cannot be foreseen at design time for all future stakeholders in an open system.

In ASEME, the inter-agent control model is a statechart itself, and the ASEME integrated development environment (IDE) allows to generate code and store it in its own package. However, this feature is not connected with the intra-agent control model generation and even though code generation is automated it only generates the control code, not the action methods of the agents. Thus, in practice, all behaviours action codes must be rewritten for each agent. This, however, poses certain risks. Protocols intended flow may be disrupted by the code of programmers, or the same code must be rewritten in several packages.

Thus, while working on this project, we identified the need for an extension of the ASEME methodology, and we have come to a solution regarding the previously identified risks using design patterns. We identified two distinct cases.

In the first case, we had a decision-making activity, e.g. in a negotiation module, where the objects of the negotiation are quite clear, however, there are different strategies for the agents to employ. We would like to allow the diverse agent developers to develop their own strategies. Thus, a specific functionality with clear parameters needs to be able to be supplied by the developing team: that is, the developers need to associate a specific algorithm to a specific activity (or JADE behaviour's action method).

Consider now the case of *SR agents*: it is clear that a service provider gets a request, processes it and then replies with an appropriate response. The processing activity, however, can be very different and complicated, and change not only based on policy, but also based on the agent's data structures and architecture. Thus, the agent now needs to define a new capability. This time it is not just an activity's algorithm that changes, it is also the agent data structures.

4 Protocol Design Patterns

We will define the design patterns using the common template that calls for defining explicitly [24, 37]:

1. the pattern's ***name*** (appearing in the section's title).

2. the problem and the ***context*** in which it appears. We will provide this context abstractly in a way that the pattern can apply in any statechart's based development method. We will also provide information for JADE developpers.
3. the ***solution***, which can be the objects, interfaces, and other information.
4. the ***results*** of the pattern's application and possible tradeoffs.
5. the ***example*** of the pattern's application.

4.1 The Functionality Pattern

Context: The developers need to associate a specific algorithm to a protocol's basic state. For JADE implementations the developers want to allow for diverse implementations of the behaviour's action method.

Solution: To remedy this situation we rely on the well-known by practitioners *factory design pattern* [19]. According to this pattern, we use a factory method that returns an instance of a method respecting an API which can be dynamically selected, even at run-time. This can be extremely useful in emergency situations where the typical algorithm fails. It can also be used for the "self-healing" of an autonomous vehicle agent.

Figure 2 illustrates the class diagram of this design pattern. The abstract class *Template1*, contains a standard and generic API that the various subclasses (e.g. *Instance1, Instance2,..., InstanceN*) use, providing their own implementation for abstract methods. This template can be the utility function used by the *EV agent* to select recommendations. The Agent (*EV*) calls the static factory method (that instantiates one of the instances) of the *Factory1* class providing the full class name of the instance it wants to use. The factory is responsible for creating the instance and providing it to the Agent. The various agent behaviours which have access to the Agent's class data, have the necessary access to utilize the algorithm according to their needs.

Results: Diverse implementations of the basic state implementation can be employed by the agent developers. The JADE action method can be automati-

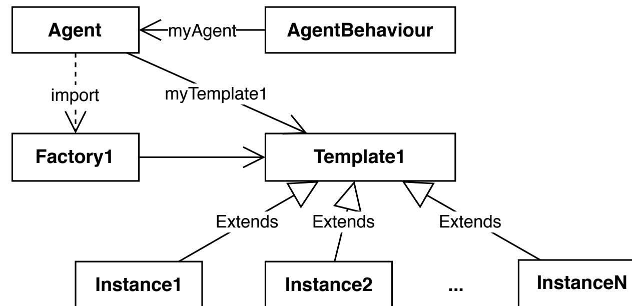

Fig. 2. The Functionality Design Pattern abstraction.

cally generated with the Template1. The developers need only specify the API of the method(s).

Example: As an example, we provide the *NegotiationProtocol* inter-agent control definition in the form of a statechart, which is presented in Figure 3. A statechart is a finite collection of states and transitions. A state can either be a basic state (with green color in the figure) or a hierarchical state, which contains multiple sub-states [21]. Now, a hierarchical state can be either an AND-state (with light blue color), or an OR-state (with yellow-colored header). AND-states have orthogonal components that are related by "and" (executed in parallel), while OR-states have sub-states that are related to each other by "exclusive-or" (only one is active at any given time). Small round states with the "c" character represent condition states where execution flow diverges based on the transition expressions of the outgoing transitions. Two more state types, the START (black dot) and END (black dot within a circle) states, indicate the start and end of execution respectively. The state at the highest level, i.e., the one with no parent state, is called the root.

Each transition from one state (source) to another (target) is labeled by an expression, 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. Transitions are binary relations between states and are usually triggered by events. Such events can be [39]:

- A sent or received inter-agent message
- A change in one of the variables of the executing state
- A timeout
- The ending of the executing state activity

A message event is expressed by $P(x, y, c)$ where P is the performative, x is the sender role, y the receiver role and c the content of the message. The items that the designer can use for defining the state transition expressions are the message performatives, the ontology used for defining the messages content and the timers. An agent can define timers as normal variables initializing them to a value representing the number of milliseconds until they timeout.

The *NegotiationProtocol* is used from *EV* and *Charging Station* agents to negotiate about an existing charging reservation. In this protocol, we have only one role, i.e. the *Responder* to a proposal. Both agents play the same role. The protocol starts as soon as a protocol-related message arrives.

The state *ReceiveMessage*, contains an activity that waits to receive a message. As soon as one such arrives, there are two possibilites:

1. if the performative is *Accept* or *t1* has timed-out, thus indicating that there is no response of the counterpart (implying inability to reach a solution), then the protocol terminates for this role (the negotiation ends unsuccessfully)
2. if the performative is *Propose* or *Reject*, then we have a transition to the *NegotiationDM* state.

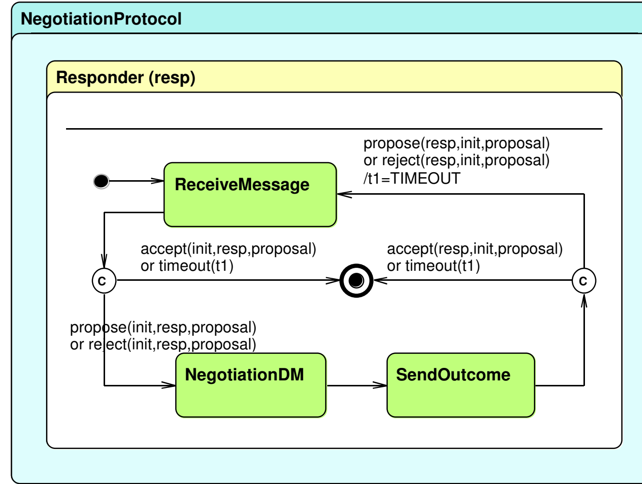

Fig. 3. The Charging Negotiation Protocol.

In the *NegotiationDM* state, the message received is processed and a result is reached based on the negotiation decision making algorithm. This algorithm can be inserted with the functionality design pattern, i.e. the implementation of its `action` method just calls the *Template1* method (see Figure 2).

4.2 The Capability Pattern

Context: The developers need to develop an activity that will take place within a relevant basic state in the protocol. This time they do not just want to associate the state with a specific algorithm. This state activity may involve complex tasks and a whole statechart might be its implementation.

For JADE developers the issue is that this state's activity is not implemented in the protocol package. Its contents are unknown, they do not want to bias future agent instances developers.

Solution: This pattern composes activities into tree-like structures (the data structure to represent the statechart is a tree structure [40]) to represent part-whole hierarchies, following the composite pattern paradigm [19]. Put differently, during the design phase, the intra-agent control model developer may change the basic state type to an OR-state type and add sub-states as they see fit, even other protocols.

For JADE code generators, this state is passed as a parameter. Thus, although all agents use the same protocol package, they each develop their own specific state activity, to which we will refer to as the *handler behaviour*. The only requirement for this design pattern to work, is the handler behaviour to have access to the agent's data structures, and the data structures of the protocol itself. Listing 1.1 shows how an Xpand technology [16] code generation template could be defined for the Eclipse platform (following the ASEME IDE).

Similar templates can be defined for other object-oriented implementations of statechart execution engines.

Listing 1.1. Capability Pattern skeleton

```
public class <ProtocolRoleName> extends FSMBehaviour {

  /*auto-generated state String representations should go here,
  such as the following line*/
  private final String HANDLER = "handler";

  public <ProtocolRoleName>(Agent a, Behaviour handler,
  DataStore ds){
    setDataStore(ds);
    handler.setDataStore(getDataStore());

    /*auto-generated transitions registrations should go here*/

    /*auto-generated state registrations should go here,
    such as the following line*/
    registerState(handler, HANDLER);
  }
}
```

Results: This allows agent implementations of different protocols to apply to completely different scale, i.e. by substituting a basic state in the statechart definition with an OR-state and define a complex behaviour.

Example: See for example the *ChargingRecommendationProtocol* definition in Figure 4. The roles of the protocol are exactly the agents that implement it. For the EV role, the protocol starts by sending a message to *Request* charging recommendations. The sender is the EV agent, the receiver is the *SR* agent, and the message content is a *requestRecommendations* instance (the properties of this action are defined in the ontology package). Then, EV goes to the *ReceiveRecommendations* state to wait for the *SR*'s results. The *SR* role is always waiting to receive requests, then it enters the *CalculateRecommendations* state to find the best options for the EV and then it enters the *SendRecommendation* state where it sends its reply.

The *CalculateRecommendations* is not implemented in the protocol package, but expected as a parameter in the instantiation of the protocol role. Thus, though all agents use the same protocol package, they each develop their own handler behaviour.

4.3 The Design Patterns Use for Developing a Real-World System

In this section we discuss our implementation, along with use cases that demonstrate the effectiveness of our architecture and design patterns in addressing the real-world needs of the various actors.

The intra-agent control model for the Electric Vehicle (EV) agent is shown in Figure 5 (to improve visibility we omit the transition expressions). Note that for simplicity of representation, the protocol roles that the agent realizes are shown as basic states. These can be expanded to the relevant roles in the protocols presented earlier in Figures 3 and 4.

When it starts its operation, the agent performs initialization activities (enters the *Init* state). Then, the agent enters both the *Negotiate* and *Reserve* OR-states as they are the orthogonal components of the *Main* AND-state. In the *Reserve* state it makes a transition to the *DecideNextAction* basic state.

In this state, the agent makes decisions regarding the charging of the EV. There are three different possibilities for the *EV agent*: *(a)* it uses a specific algorithm to monitor and predict the battery state and driver preferences, and to decide autonomously when and how to arrange the EV charging; *(b)* it gets the aforementioned information from predefined datasets;[4] and *(c)* it provides a comprehensive GUI where the user can insert her charging preferences and manually initiate the protocols. These possibilities are three different implementations for the *DecideNextAction* state activity.

[4] This is very useful for experimentation with large agent populations.

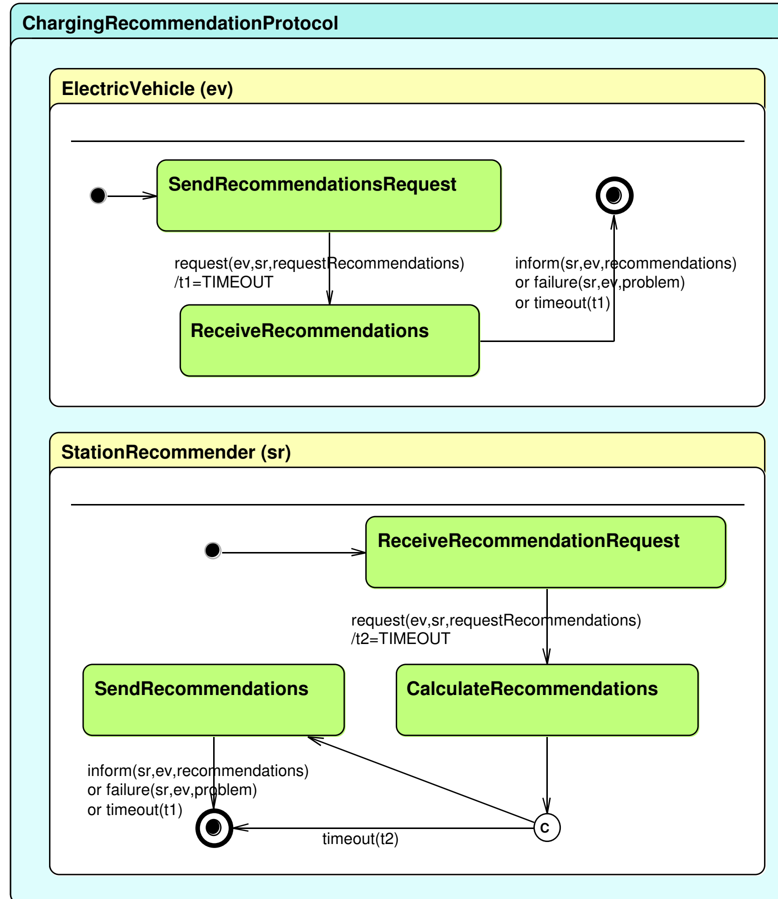

Fig. 4. The Charging Recommendation Protocol.

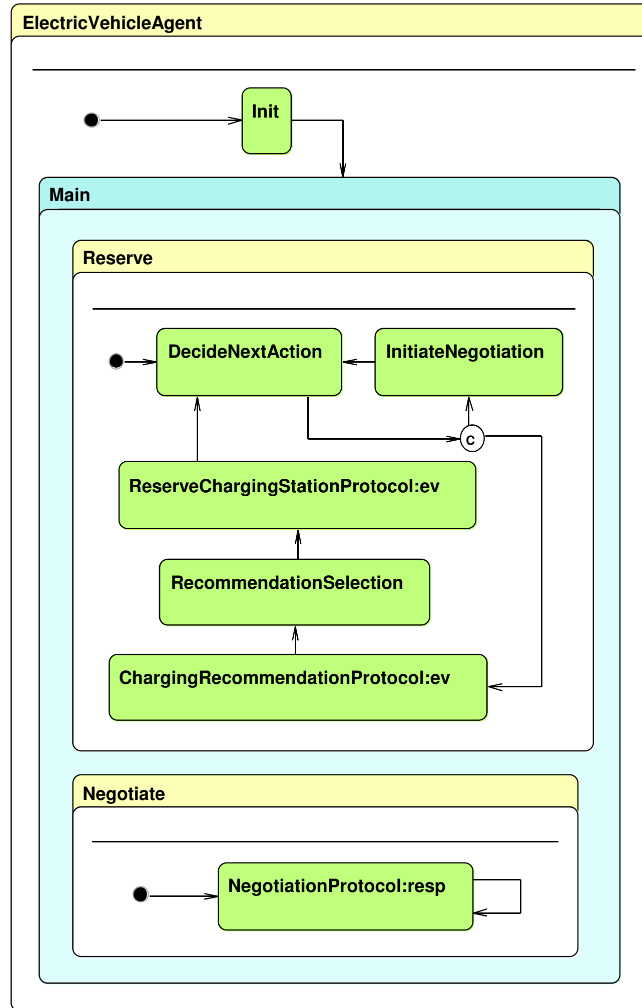

Fig. 5. The EV agent intra-agent control model. Transition expressions have been omitted so as not to clutter the diagram.

Whenever the agent decides to arrange a next charging, it goes to the *ChargingRecommendationProtocol:ev* state (using the *ev* role of the respective protocol), then to the *RecommendationSelection* state (corresponding to the agent functionality with the same name—see Figure 1), and, finally to the *ReserveChargingStationProtocol:ev* to reserve the selected slot. Then, it returns to the *DecideNextAction* state from where it will now leave in order to make a new reservation or to negotiate a change in an existing arrangement using the *InitiateNegotiation* state. The latter just sends a proposal, and if the *CS* agent replies, the rest of the negotiation process will be taken care by the *NegotiationProtocol:resp*

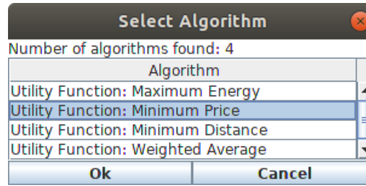

Fig. 6. Different algorithms for selecting the best station. The user selects the desired one at runtime, or the agent selects it after assessing user preferences.

state (using the *responder* role of the respective protocol) at the *Negotiate* component. This is executed always in parallel to the *Reserve* component, as a *CS* agent could itself initiate a negotiation at any time.

The *EV* agent developer used the functionality design pattern to define the *NegotiationDM* action of the *NegotiationProtocol* (see Figure 3). The same pattern has been used for other basic states allowing for diverse implementations. In the specific case of the *RecommendationSelection* state we wanted to give different options to the user and we provided four different implementations ourselves and the user can select the one they prefer through a GUI (see Figure 6).

In Figure 7 the reader can see part of the Station Recommender (SR) agent intra-agent control. This part shows the initialization of its behaviour and then its role of the *Charging Recommendation Protocol* being executed in parallel with other capabilities of the agent. Note that as soon as a recommendation is sent the agent restarts the protocol role (see the transition from the *ChargingRecommendationProtocol:sr* state to itself).

Here you can note the application of the Capability pattern as the SR agent extends the original *CalculateRecommendations* basic state (activity) of the protocol to an OR-state invoking two more protocols (the *ElectricityImbalanceRequestProtocol* with the role *ServiceUser* and the *ElectricityPricesRequestProtocol* with the role *ServiceUser*) and then doing a calculation (*CalculateChargingRecommendations* state). You can also note that since many agents request for imbalance or prices the relevant protocols use abstract roles (e.g. *ServiceUser*) and concrete roles such as the SR agent implement these protocols by assuming those abstract roles.

5 Conclusions and Future Work

It is not the first time that the need for design patterns is identified in the MAS development literature. Early researchers identified the need for patterns in the collaboration layer [24, 37] and focused in the communication part of the protocols leading to the first patterns that were applied to methodologies (e.g. PASSI [13]) and accommodated the use of specific protocol patterns that were identified by FIPA (e.g. *request*, *contract-net*, etc). Such patterns are used in the high methodological level and are implemented through existing support in platforms (e.g. JADE [6, 5]).

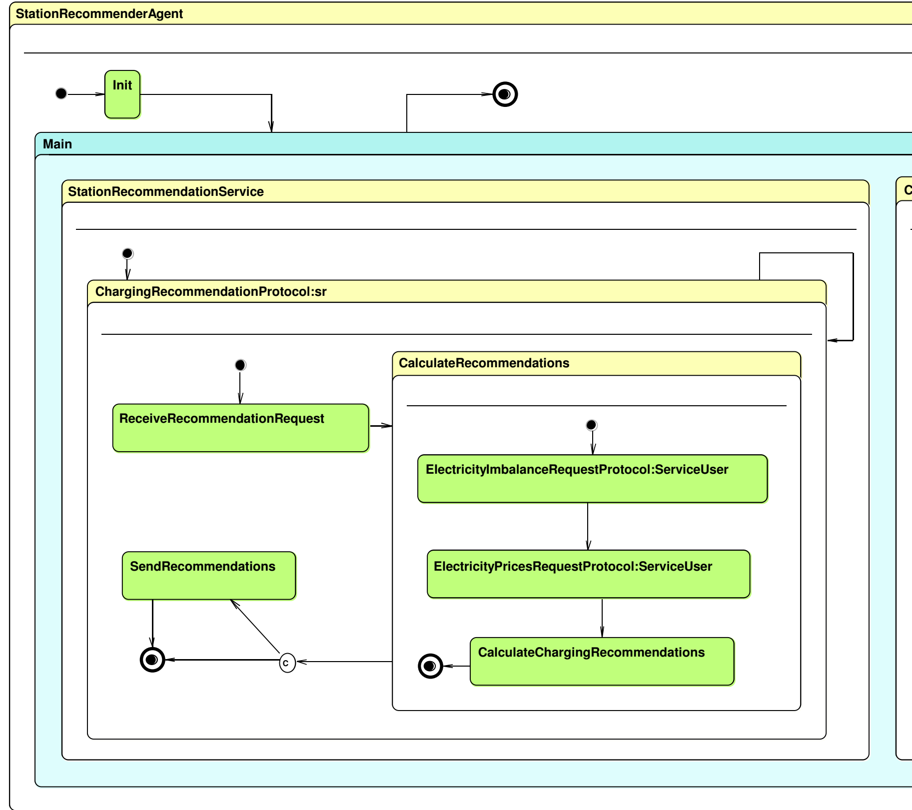

Fig. 7. Part of the SR agent intra-agent control model.

In this paper we put forward and implemented two novel design patterns, allowing for participating agents *(a)* to dynamically select functionalities and *(b)* to define their own implementations of abstract behaviours. Moreover, we demonstrated the use of these patterns within a novel MAS architecture for the V2G/G2V energy transfer problem domain, for which we have developed a fully functional prototype. Our approach addresses the needs for openness, and the coverage of diverse business models via the definition of a number of key agent types and the development of open protocols. These can be delivered along with the ontology to any interested parties, which can subsequently build their own agents given their expertise and business case.

These patterns are agent methodology- and platform-independent. They can be applied with any statecharts-based method and influence code generation for diverse platforms. Both patterns are compatible with object-oriented development, which is supported by many methodologies for Engineering MAS (e.g. [2, 11, 14, 30]).

References

1. Al-Agtash, S., Hafez, H.A.: Agents for smart power grids. Energy and Power Engineering **12**(08), 477–489 (2020)
2. Alaca, O.F., Tezel, B.T., Challenger, M., Goulão, M., Amaral, V., Kardas, G.: AgentDSM-Eval: A framework for the evaluation of domain-specific modeling languages for multi-agent systems. Computer Standards & Interfaces **76**, 103513 (2021). https://doi.org/10.1016/j.csi.2021.103513
3. Asmus, P.: Microgrids, virtual power plants and our distributed energy future. The Electricity Journal **23**(10), 72 – 82 (2010)
4. Azeroual, M., Lamhamdi, T., El Moussaoui, H., El Markhi, H.: Simulation tools for a smart grid and energy management for microgrid with wind power using multi-agent system. Wind Engineering **44**(6), 661–672 (2020). https://doi.org/10.1177/0309524X19862755
5. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology). John Wiley and Sons Ltd (2007)
6. Bellifemine, F., Poggi, A., Rimassa, G.: Developing multi-agent systems with a fipa-compliant agent framework. Software: Practice and Experience **31**(2), 103–128 (2001)
7. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible bdi agent modularization. In: International Workshop on Programming Multi-Agent Systems. pp. 139–155. Springer (2005)
8. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems **8**(3), 203–236 (2004). https://doi.org/10.1023/B:AGNT.0000018806.20944.ef
9. Burke, M.J., Stephens, J.C.: Energy democracy: Goals and policy instruments for sociotechnical transitions. Energy Research & Social Science **33**, 35 – 48 (2017), policy mixes for energy transitions
10. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring bdi agents in functional clusters. In: International Workshop on Agent Theories, Architectures, and Languages. pp. 277–289. Springer (1999)
11. Cossentino, M., Seidita, V.: Passi: Process for agent societies specification and implementation. In: Handbook on Agent-Oriented Design Processes, pp. 287–329. Springer (2014)
12. Cossentino, M.: From Requirements to Code with the PASSI Methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, chap. IV, pp. 79–106. Idea Group Publishing (2005)
13. Cossentino, M., Sabatucci, L., Chella, A.: Patterns reuse in the passi methodology. In: Omicini, A., Petta, P., Pitt, J. (eds.) Engineering Societies in the Agents World IV. pp. 294–310. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25946-6_19
14. DeLoach, S.A., Garcia-Ojeda, J.C.: O-MaSE: a customisable approach to designing and building complex, adaptive multi-agent systems. International Journal of Agent-Oriented Software Engineering **4**(3), 244–280 (2010)
15. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent systems engineering. International Journal of Software Engineering and Knowledge Engineering **11**(03), 231–258 (2001)
16. Efftinge, S., Völter, M.: oaw xtext: A framework for textual dsls. In: Workshop on Modeling Symposium at Eclipse Summit. vol. 32, p. 118 (2006)

17. Espe, E., Potdar, V., Chang, E.: Prosumer communities and relationships in smart grids: A literature review, evolution and future directions. Energies **11**(10), 2528 (2018)
18. Feliachi, A., Belkacemi, R.: Intelligent multi-agent system for smart grid power management. In: Smart Power Grids 2011, pp. 515–542. Springer (2011)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Reading, Mass (1995)
20. Garijo, F.J., Gomez-Sanz, J.J., Massonet, P.: The MESSAGE Methodology for Agent-Oriented Analysis and Design. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, chap. VIII, pp. 203–235. Idea Group Publishing (2005)
21. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology (TOSEM) **5**(4), 293 (1996)
22. Huynh, T.D., Jennings, N.R., Shadbolt, N.R.: An integrated trust and reputation model for open multi-agent systems. Autonomous Agents and Multi-Agent Systems **13**(2), 119–154 (2006)
23. Iglesias, C.A., Garijo, M.: The Agent-Oriented Methodology MAS-CommonKADS. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, chap. III, pp. 46–78. Idea Group Publishing (2005)
24. Kendall, E.A., Krishna, P.V.M., Pathak, C.V., Suresh, C.B.: Patterns of intelligent and mobile agents. In: Proceedings of the Second International Conference on Autonomous Agents. pp. 92–99. AGENTS '98, ACM, New York, NY, USA (1998)
25. Ketter, W., Collins, J., Reddy, P.: Power tac: A competitive economic simulation of the smart grid. Energy Economics **39**, 262 – 270 (2013)
26. Kuzin, A.Y., Demidova, G.L., Lukichev, D.V.: An approach of the jade and simulink interaction to control smart grid based on the multi agent system. In: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus). pp. 574–577. IEEE (2019)
27. Mazouzi, H., Seghrouchni, A.E.F., Haddad, S.: Open protocol design for complex interactions in multi-agent systems. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2. pp. 517–526. AAMAS '02, ACM, New York, NY, USA (2002)
28. Moore, S.A.: On Conversation Policies and the Need for Exceptions. In: Dignum, F., Greaves, M. (eds.) Issues in Agent Communication, Lecture Notes in Computer Science, vol. 1916, pp. 144–159. Springer, Heidelberg (2000). https://doi.org/10.1007/10722777
29. Murray, J.: Specifying agent behaviors with uml statecharts and statedit. In: Polani, D., Browning, B., Bonarini, A., Yoshida, K. (eds.) RoboCup 2003: Robot Soccer World Cup VII. pp. 145–156. Springer Berlin Heidelberg (2004)
30. Odell, J., Parunak, H.V.D., Bauer, B.: Extending uml for agents. In: Proceedings of the agent-oriented information systems workshop at the 17th national conference on artificial intelligence. pp. 3–17 (2000)
31. Odell, J.J., Parunak, H.V.D., Bauer, B.: Representing agent interaction protocols in uml. In: International Workshop on Agent-Oriented Software Engineering. pp. 121–140. Springer (2000)
32. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. Wiley Series in Agent Technology, Wiley (2004)
33. 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.) Programming Multi-Agent Systems. pp. 149–168. Springer Berlin Heidelberg (2004)

34. Picard, G., Gleizes, M.P.: The ADELFE Methodology. In: Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.) Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook, pp. 157–175. Springer US, Boston, MA (2004). https://doi.org/10.1007/1-4020-8058-1_11
35. Priyadarshana, H.V.V., U Hemapala, K.T.M., S Wijayapala, W.D.A., Saravanan, V., Kalhan S Boralessa, M.A.: Developing Multi-Agent Based Micro-Grid Management System in JADE. In: 2nd International Conference on Power and Embedded Drive Control (ICPEDC). pp. 552–556. IEEE (2019)
36. Ramchurn, S.D., Vytelingum, P., Rogers, A., Jennings, N.R.: Putting the 'smarts' into the smart grid: a grand challenge for artificial intelligence. Commun. ACM **55**(4), 86–97 (2012)
37. Sauvage, S.: Design patterns for multiagent systems design. In: Monroy, R., Arroyo-Figueroa, G., Sucar, L.E., Sossa, H. (eds.) MICAI 2004: Advances in Artificial Intelligence. pp. 352–361. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
38. Shobole, A.A., Wadi, M.: Multiagent systems application for the smart grid protection. Renewable and Sustainable Energy Reviews **149**, 111352 (2021). https://doi.org/10.1016/j.rser.2021.111352
39. 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-Volume 02. pp. 578–582. IEEE Computer Society (2008)
40. Spanoudakis, N.I., Moraitis, P.: Gaia Agents Implementation through Models Transformation. In: Principles of Practice in Multi-Agent Systems, 12th International Conference, PRIMA 2009, Nagoya, Japan, December 14-16, 2009. Proceedings. pp. 127–142 (2009)
41. Spanoudakis, N.I., Moraitis, P.: Modular JADE Agents Design and Implementation Using ASEME. In: Proceedings of the 2010 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2010, Toronto, Canada, August 31 - September 3, 2010. pp. 221–228 (2010)
42. Spanoudakis, N., Akasiadis, C., Kechagias, G., Chalkiadakis, G.: An open MAS services architecture for the V2G/G2V problem. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. pp. 2198–2200 (2019)
43. Spanoudakis, N.I.: Engineering multi-agent systems with statecharts: Theory and practice. SN Computer Science **2**(4), 317 (2021). https://doi.org/10.1007/s42979-021-00706-5
44. Spanoudakis, N.I., Akasiadis, C., Iatrakis, G., Chalkiadakis, G.: Engineering IoT-Based Open MAS for Large-Scale V2G/G2V. Systems **11**(3), 157 (2023)
45. Spanoudakis, N.I., Moraitis, P.: The ASEME Methodology. International Journal of Agent-Oriented Software Engineering **7**(2), 79–107 (2022)
46. Trencansky, I., Cervenka, R.: Agent Modeling Language (AML): a comprehensive approach to modeling MAS. Informatica (Slovenia) **29**(4), 391–400 (nov 2005)
47. Wooldridge, M., Jennings, N.R.: Intelligent agents: Theory and practice. Knowledge Engineering Review **10**, 115–152 (1995)
48. Wooldridge, M.J.: Introduction to multiagent systems. John Wiley and Sons Ltd (2002)