Application Framework with Abstractions for Protocol and Agent Role

Kristensen, Bent Bruun

Published in:
Proceedings of the Fourth International Workshop on Engineering Multi-Agent Systems

DOI:
10.1007/978-3-319-50983-9_6

Publication date:
2016

Document version
Version created as part of publication process; publisher's layout; not normally made publicly available

Citation for published version (APA):
https://doi.org/10.1007/978-3-319-50983-9_6
Abstract. In multi-agent systems, agents interact by sending and receiving messages and the actual sequences of message form interaction structures between agents. Protocols and agents organized internally by agent roles support these interaction structures. Description and use of protocols based on agent roles are supported by a simple and expressive application framework.

Keywords: Multi-agent system, Protocol, Agent role, Reactive and proactive role, Application framework.

1 Introduction

Agents are active, autonomous, and smart, i.e. among others capable of reactive and pro-active behavior [1]. A multi-agent system consists of a number of agents interacting with one-another— and to successfully interact, agents require the ability to cooperate, coordinate, and negotiate with each other. We describe interactions by protocols that relate agent roles of such communicating agents.

Abstraction is essential: “Without abstraction we only know that everything is different” [2] meaning that use of abstractions to describe observations is essential for the resulting understanding. Dialogues and agent roles are abstractions, i.e. by these concepts the developer can understand and describe the organization structure of agents as well as the interaction structure between communicating agents.

Our aim is to support protocols that capture communication between agent roles by an object-oriented application framework [3] with agents, reactive role, proactive role and message. The underlying agent model and the application framework are illustrated by the Contract Net [1]. The model and framework are compared to related work and evaluated.

2 Agent Model

Reactive and Proactive Roles. Agents communicate by sending and receiving messages representing events as illustrated in Fig. 1. An agent consists of a varying number of reactive and proactive roles. Reactive and proactive roles are abstractions for internal organization of an agent and messages are sent from and received by these
roles. If a message is sent to the agent itself a default reactive role of the agent receives the message.

The roles of an agent execute one at a time and in a non preemptive way [4], i.e. they exhibit cooperative multitasking, in which case a role can self-interrupt and voluntarily give up control. Reactive and proactive roles are stereotypes but combinations can be described. Each reactive and proactive role has a list of messages to be handled on a first come first served basis. A reactive role repeats the execution of an action to take care of its list of messages whenever the handling of the previous message is completed and the awaiting message list is not empty. A proactive role consists of a single execution of an action that takes care of pausing as well as waiting and handling messages until its purpose is completed.

![Fig. 1. Agents organized by reactive and proactive roles.](image1)

**Protocols.** A protocol describes a process where an initiator initializes the interaction by sending messages to a number of participants where after these participants may reply to the initiator as part of the interaction, etc. The protocol takes place between proactive roles of agents. The protocol and the proactive roles together form abstractions over an interaction structure between the involved agents.

![Fig. 2. Illustration of protocol and proactive roles.](image2)

Fig. 2 illustrates a protocol \( P \) between proactive roles \( R_1 \) and \( R_2 \) in agents \( A_1 \) and \( A_2 \). Role \( R_1 \) initializes the interaction by sending a message \( M_1 \) to role \( R_2 \). Role \( R_2 \) replies with message \( M_2 \) to \( R_1 \). In this manner a protocol between \( R_1 \) and \( R_2 \) may describe a continued interaction between \( R_1 \) (from \( A_1 \)) and \( R_2 \) (from \( A_2 \)). The protocol illustrated in Fig. 2 is similar to the coroutine mechanism of SIMULA [5] in the sense that a role sends a message, immediately suspends itself and the receiving role is resumed.
Example: Contract Net. Fig. 3 illustrates the Contract Net with a collection of stickmen. Each stickman in the collection can, at different times or for different tasks, be involved in several simultaneous tasks as both manager and contractor. When a stickman gets a composite task (or for any reason cannot solve its present task), it breaks the task into subtasks (if possible) and announces them (acting as a manager), receives bids from potential contractors, and then possibly awards a contractor. If no bids are received after a given period of time the manager gives up the negotiation. If a bid is not awarded after a given period the contractor gives up the negotiation.

Fig. 4. Conceptual Model showing the contents of and relations between Agent System, Utility System and Test System.
A model for the Contract Net includes: A protocol is set up with a proactive role for the manager agent (the initiator) and a proactive role for each of the contractor agents (the participants). A manager maintains a negotiation by initiating an interaction with a number of contractors. A contractor receives a task announcement and may reply with a bid to the manager. Having received bids the manager chooses among these and may reply with an award to the chosen contractor in which case a contract is established.

3 Framework Overview

Fig. 4 illustrates the conceptual model of the application framework with Agent_System that is specialized to another application framework Utility_System (to support various protocols with the Contract Net as an example) that in turn is used in the application Test_System. The contents of and relations between Agent_System, Utility_System and Test_System are described in the following sections.

... class Agent_System {
  ...
  abstract class Protocol {
    ...
    init initiatorRole = initiator.newInitiatorRole(this);
    for (int i = 0; i < n; i++) {
      participantRole[i] = participant[i].newParticipantRole();
    }
  }
  ...
  Agent.Initiator initiatorRole;
  Agent.Participant[] participantRole = ...
}

... abstract class Agent extends .. implements .. {
  ...
  abstract class Agent_Role extends .. {
    ...
    abstract class ReActive_Role extends Agent_Role {
      abstract int Act(Agent_Message am)
    };
    ...
    abstract class ProActive_Role extends Agent_Role {
      abstract void Act()
    };
    ...
  ...
  abstract class Initiator extends ProActive_Role {
    ...
  ...
  abstract class Participant extends ProActive_Role {
    ...
  }
  ...
  abstract class Agent_Message extends EventObject {...
}

Fig. 5. Application Framework: Agent_System
4 Application Framework: Agent System

Fig. 5, 6 and 7 show extracts of the textual version of the application framework Agent_System with classes and methods shown in grey. Fig. 5 shows class Agent_System with abstract classes Agent, Protocol and Agent_Message. Class Agent has abstract classes ReActive_Role and ProActive_Role (extending class Agent_Role). Class ProActive_Role is extended to classes Initiator and Participant both related to class Protocol. In addition classes ReActive_Role and ProActive_Role include the abstract method Act(…). The Act(…) method of class ReActive_Role returns a delay until next invocation and is invoked repeatedly with the next message as parameter while messages are waiting. The Act() method of class ProActive_Role is invoked only once and its execution may include pausing, awaiting messages, etc. until its execution is completed. Abstract methods newInitiatorRole and newParticipantRole are used by Protocol to instruct actual specializations of Agent to instantiate actual specializations of Initiator and Participant. Class Protocol instantiates and starts the execution of InitiatorRole (with the Protocol object as parameter) for InitiatorAgent and of ParticipantRole for each of the ParticipantAgents.

Fig. 6. Interaction methods of class Agent_Role

Fig. 6 shows extracts of selected interaction methods of class Agent_Role (inherited by ReActive_Role and ProActive_Role):

- rolePause(…), the role pauses for a period of time
- roleAwait(), the role waits until a message is received and then returns the message
- replyMessage(…), a message is sent to a role of another agent as a reply to a message received from that role
- handleMessage(), the next waiting message is returned (if any and else null)

Fig. 7. Methods initiateProtocol and replyProtocol
Class Protocol sets up the protocol between agent roles—the Initiator and the Participants agents. Fig. 7 shows extracts of the interaction methods related to the protocol:

- initiateProtocol(...), the Initiator sends a message \( am \) to a Participant to initialize the protocol.
- replyProtocol(...), either the Initiator or a Participant send a message \( ram \) in reply to message \( am \) received within the protocol.

5 Application Framework: Utility System

Class Agent_System can be used directly to construct a multi-agent system with reactive and proactive agent roles and protocols. We choose to extend Agent_System to another abstract class Utility_System to illustrate an example of an abstract protocol—the Contract Net. Classes Utility_System is then specialized in class Test_System as an actual use.

Fig. 8 shows the ingredients of the specialization of Agent_System to Utility_System: Protocol is specialized to CN_Protocol and the proactive roles Initiator and Participant to Manager and Contractor, respectively. Classes Manager and Contractor specify their own Act() method according to the Contract Net. And each of these Act() methods makes use of additional methods (shown as dotted) to be implemented in the actual use of the Utility_System.

![Fig. 8. Specialization of Agent_System to Utility_System](image)

Fig. 9 illustrates how Utility_System and Utility_Agent extend Agent_System and Agent, respectively. Class Protocol is extended to CN_Protocol that initializes a Manager role for the Initiator agent and a Contractor role for each of the Participant agents.
abstract class Utility_System extends Agent_System {
    
    class CN_Protocol extends Protocol {
        CN_Protocol (Agent initiator, Agent[] participant) {
            super (initiator, participant);
            
            Utility_Agent.Manager managerRole;
            Utility_Agent.Contractor[] contractorRole = ... ;
        }
    }

    abstract class Utility_Agent extends Agent {
        abstract class Manager extends Initiator {
            Manager (Protocol protocol) {};
            
            void Act() {
                ...
            }
        }
        
        abstract class Contractor extends Participant {
            void Act() {
                ...
            }
        }
    }
}

Fig. 9. Utility_System with CN_Protocol

When a Protocol is instantiated as shown in Fig. 10 its constructor initializes ManagerRole through the executing agent and ContractorRoles through otherAgents.

new CN_Protocol(Test_Agent.this, otherAgents);

Fig. 10: Creation of a CN_Protocol

Fig. 11 shows abstract class Manager as an extension of Initiator. Method Act() of class Manager uses the abstract methods (hot spots cf. [6]) in italics (implemented in Test_System).

abstract class Manager extends Initiator {
    
    void Act() {
        ...
    }

    abstract CN_Task createCN_Task(int t)
    abstract CN_Task.Offer createOffer(CN_Task t)
    abstract CN_Message newOfferMessage(Agent a, CN_Task t)
    abstract int bidDelayTime()
    abstract CN_Message selectBid(CN_Message[] ms, int l)
    abstract CN_Task.Award createAward(CN_Task t)
    abstract CN_Message newAwardMessage(Agent a, CN_Task t)
}

Fig. 11. Class Manager

Fig. 12 shows the actual sequencing in the Act() method of class Manager—illustrated by the comments: Prepare and send offers, Wait a while until
bids have arrived, Collect received bids, Select a bid and prepare and send an award.

```java
void Act() {
    CN_Protocol cnd = (CN_Protocol) protocol;
    CN_Message m;
    // ... ... Prepare and send offers ...
    int taskNo = allTasks++;
    for (int i = 0; i < cnd.contractorAgent.length; i++) {
        CN_Task t = createCN_Task(taskNo);
        if (t.addOffer(createOffer(t))) {
            m = newOfferMessage(cnd.contractorAgent[i], t);
            initiateProtocol(m, cnd.contractorRole[i]);
        }
    }
    // ... ... Wait a while until bids have arrived ...
    rolePause(bidDelayTime());
    // ... ... Collect received bids ...
    CN_Message[] ms = new CN_Message[...];
    int i = 0;
    while ((m = (CN_Message) handleMessage()) != null) {
        if (m.typeMessage(CN_Kind.BID)) {
            ms[i++] = m;
        }
    }
    // ... ... Select a bid, prepare and send an award ...
    if ((m = selectBid(ms, i)) != null) {
        CN_Task t = m.cnTask;
        t.addAward(createAward(t));
        CN_Message mm = newAwardMessage(m.fromAgent, t);
        replyProtocol(mm, m);
    }
}
```

**Fig. 12. Method Act() of Manager**

Fig. 13 shows abstract class Contractor as an extension of Participant. Method Act() of class Contractor uses the abstract methods in italics (implemented in Test_System).

```java
abstract class Contractor extends Participant {
    // ...
    void Act() { ...

    abstract CN_Task.Bid createBid(CN_Task t)
    abstract CN_Message newBidMessage(Agent a, CN_Task t)
    abstract int awardDelayTime()
    abstract void handleAward(CN_Message m)
}
```

**Fig. 13. Class Contractor**

Fig. 14 shows the actual sequencing in the Act() method of class Contractor—illustrated by the comments: Wait to receive an offer, Possibly prepare and send a bid, Wait to receive an award, Possibly receive and handle the award.
```java
... void Act() {
    // ... Wait to receive an offer ...  
    CN_Message m = (CN_Message) roleAwait();
    if (m.typeMessage(CN_Kind.OFFER)) {
        // ... Possibly prepare and send a bid ...  
        CN_Task t = m.cnTask;
        if (t.addBid(createBid(t))) {
            CN_Message mm = newBidMessage(m.fromAgent, t);
            replyProtocol(mm, m);
        }  
        // ... Wait to receive an award ...  
        rolePause(awardDelayTime());
        // ... Possibly receive and handle the award ...  
        if (m = (CN_Message) handleMessage() != null) {
            if (m.typeMessage(CN_Kind.AWARD)) {
                handleAward(m);
            }
        }
    }
    else return;
}
```

Fig. 14. Method Act() of Contractor

Fig. 15 shows CN_Message as an extension of Agent_Message where CN_Task represents the actual task to be undertaken (with respect to Offer, Bid and Award) and CN_Kind enumerates the actual message types in Contract Net.

```java
... class CN_Message extends Agent_Message {
    ... CN_Message(...) CN_Task cnt, CN_Type cnk, ...) {
    ... 
    ... CN_KIND cnk;
    ... CN_Task cnt;
}
... class CN_Task ...  
enum CN_KIND {OFFER, BID, AWARD}
```

Fig. 15. Classes CN_Message, CN_Task and CN_KIND

6 Test System

Fig. 16 shows Test_System, as an extension of Utility_System where class Test_Agent extends Utility_Agent. The abstract methods newInitiatorRole and newparticipantRole are implemented to return objects of the actual Manager and Contractor classes specialized from Manager and Contractor of Utility_Agent. Classes Manager and Contractor implement the abstract methods from Fig. 11 and 13, respectively.
class Test_System extends Utility_System {

class Test_Agent extends Utility_Agent {
    Initiator newInitiatorRole(Protocol protocol) {
        return (new Manager((CN_Protocol) protocol));
    }
    Participant newParticipantRole() {
        return (new Contractor());
    }
}

class Manager extends Utility_Agent.Manager {
}
class Contractor extends Utility_Agent.Contractor {
}
}

Fig. 16. Test_System with Test_Agent

The protocol using the OFFER, BID and AWARD messages is simple and therefore the structures of the roles illustrated in Fig. 12 and 14 are simple too. But these roles would remain simple if they involved additional interaction, i.e. such as re-announcing subtasks, continued negotiations about details, etc. A Test_Agent involved in several simultaneous contract negotiations would not complicate the description but only require additional instantiations of the existing protocol.

Fig. 17 is a snapshot of the dynamic flow of messages between agents. This feature is a part of the application framework, i.e. it is general although it is parameterized with the actual extension of the framework—in this case Contract Net. For each agent, i.e. for Test_Agent 2 there is a column of messages sent Messages Out: 6 and received Messages In: 6 showing total number of messages and a list of actual messages. The actual messages are colored to indicate the status of a message, i.e. sent, received, forwarded, handled and to be removed. It can be seen from Fig. 17 that TEST_Agent2 sends offer 5 that is received by TEST_Agent1; TEST_Agent1 replies with bid 5 that is received by TEST_Agent2; TEST_Agent2 replies with award 5 that is received by TEST_Agent1. This protocol is similar to the M_1, M_2, M_3 protocol illustrated in Fig. 2.

Fig. 17. Flow of messages between agents
7 Background, Related Work, Evaluation

**Background.** The FLIP project [7] investigates a transportation system including moving boxes from a conveyor belt onto pallets and transporting these pallets in the high bay area of the LEGO® factory with AGVs, no human intervention and only centralized control. A toy prototype includes agents in the form of LEGOBots based on a LEGO® MindstormsTM RCX brick extended with a PDA and wireless LAN. The DECIDE project [8] includes a number of real applications: Control of a baggage handling system in a larger airport in Asia; Intelligent control of handling material with recipes in productions processes; Coordination and planning of large vehicle transports at a shipyard; Design and implementation of a very flexible packing machine. These applications illustrates that the complexity of the communication structure between agents needs to be supported by structurally simple and expressive abstractions.

A course about agent oriented programming includes the construction of a multi-agent system based on the application framework. The task is to design and implement the management of the evolution of a collection of animal parks. A solution is to use reactive roles to react to incoming messages concerning actual changes—and proactive roles to support buying and selling animals by negotiating with other agents. The experience includes that the complexity of the communication structure of simple toy-like multi-agent systems is overwhelming, because the basic communication sequence is simple but the management of several simultaneously ongoing communication sequences is complicated.

**Related Work.** The application framework is for implementing protocols and agent roles, i.e. for describing abstractions and using these in concrete applications. The purpose of [9] is modelling of agent interaction protocols in AUML as a set of UML idioms and extensions. In [10] the purpose is to specify, validate and evaluate interaction protocols expressed as recursive colored petri nets. The purpose of [11] is to experiment with the enhancement of object orientation with agent-like interaction including protocol and role introduced in the powerJava extension of Java to allow session-aware interactions. In [12] the purpose is to load interaction protocols dynamically through role, action and message ontologies, process description with decision-making rules and a three-layer agent architecture. Dialogue games are the basis for agent interaction protocols for convincing through arguments—in [13] by formal definition of the PARMA protocol—in [14] by a categorization of types of dialogue games with examples of protocols.

The use of object-oriented languages for creating frameworks with concepts from multi-agents is well known, as well as the notion of protocol, agent role, reactive and proactive agents. But the actual form of protocol, reactive and proactive roles of agents and their inclusion in the application framework is original. A protocol is between one initiator and several participants, i.e. the initiator sends a message to the participants that may send a message back to the initiator, i.e. the initiator communicate with each of the participants but the participants do not communicate together. Protocols may be organized with part-protocols to support that a participant (as part of an ongoing protocol) may be initiator of a part-protocol.
Reactive and proactive roles are related to *behaviours* in JADE [15] and *plans* in JACK [16]: In JADE the agent life-cycle is described by behaviors by extending the `Behaviour` class. An agent can execute several behaviors in parallel. However, behavior scheduling is not preemptive, but cooperative—and everything occurs within a single Java thread. In JACK an agent will look for the appropriate plans to handle goals and events. The plan (an abstraction above object-oriented constructs) inherits from a `Plan` class that implements the plan’s base methods and the underlying functionality. Neither behaviors nor plans support the notion of reactive and proactive role explicitly but may be utilized to expose similar behavior. In JADE `createReply()` creates a new message properly setting the receivers and various fields used to control the conversation. In JACK `reply(received, sendBack)` sends a message back to an agent from which a previous message has been received without triggering a new plan.

**Evaluation.** Each of the $n$ agents in the Contract Net example may send an offer to the $n-1$ other agents that may reply back etc.:

- Without the protocol abstraction we assume each agent has one (typically reactive) role, i.e. $n$ roles in total. However such a role has to manage up to $n$ ongoing communications (one of which is between up to $n$ agents) each with their own state of the communication. Without the protocol abstraction each role takes care of $n$ communications.

- With the protocol abstraction each agent has a manager role and sends an offer to $n-1$ other agents each with a contractor role, i.e. in total $n$ roles. When $n$ agents send an offer this becomes $n^2$ roles in total. However each role is simple as illustrated in Fig. 12 and 14 because each role is involved in exactly one protocol, i.e. the state of the communication is captured by the role. With the protocol abstraction each role takes care of 1 communication.

In summary the agent model and framework are simple and understandable but still expressive. By the abstractions protocol and agent role we substitute the usual complexity of describing the handling (including state and progress) of several simultaneously ongoing communications by simple, statically structured protocols and roles.

By identifying protocol and agent roles in the Contract Net we classify the interaction and the contributions of the agents by means of `CN_protocol`, `Manager` and `Contractor`. However, abstraction includes not only classification but also specialization and composition: We may see `CN_protocol`, `Manager` and `Contractor` as a general description of the Contract Net, so that specialized versions of Contract Net can be described by specializations of each of these abstractions, e.g. `CN_protocol_X`, `Manager_X` and `Contractor_X`. Similarly, another more extensive protocol can be composed by using the Contract Net as a part protocol by using `CN_protocol`, `Manager` and `Contractor` in the description of this protocol.

Classes `Protocol`, `Initiator` and `Participant` together form abstractions over an interaction structure. `Initiator` and `Participant` are local to an `Agent` in order to have access to the local state of the `Agent`. Alternative solutions may be inspired from [17] where `Association` is a central abstraction over interaction sequences and integrate activities and roles of concurrent autonomous entities.
7 Summary

Typically the communication structure between agents becomes complicated because powerful abstractions are not available for modelling and programming. The application framework with protocols based on agent roles of agents offer abstract description of simple and expressive multi-agent communication structures.

Acknowledgments. We thank Palle Nowack and Daniel May for inspiration and support.

References