3

Scenario-Based Modeling of Multi-Agent Systems

Armin Stranjak[1], Igor Čavrak[2] and Mario Žagar[2]
[1]*Software Centre of Excellence, Rolls-Royce Plc., Derby*
[2]*University of Zagreb, Faculty of Electrical Engineering and Computing*
[1]*United Kingdom*
[2]*Croatia*

1. Introduction

Constant expansion of network-centric services inevitably led into development of new software technologies that would enable seamless and transparent access to the expanding amount of information. The software agent-oriented model fits convincingly well within this context as a better-suited technology over typical modular, client-server approach. This model offers concepts like autonomous behavior, competitive or collaborative interactions, and seamless integration with the legacy systems (Jennings et al., 1999). Based on their capabilities and predefined and/or accumulated knowledge, agents can react to changes by adapting to the new circumstances if a better approach is identified. This results in a dynamic system, well suited to coping with an ever-changing environment. Agents individually choose to co-operate or compete with other agents in order to satisfy their own objectives, but by setting goals appropriately, their collective behavior can be engineered to achieve global system-wide objectives. This approach is an efficient way of handling the complexity of many modern software systems.

Within the competitive market context, agents interact with each other in order to win access to the shared resources, to get a better price or to bet for more processing power, etc. while trying to fulfill their plans by achieving given goals. Alternatively, cooperative environment promotes such agents that will perform their goals in the interest of the wider community or the authoritative entity that secures the fulfillment of the global goal. Typical examples would be applications for task planning and resource scheduling, search engines, or any other where the emergent behavior is influenced by collaborative and mutually non-exclusive individual goals.

Regardless of the environment characteristics used, agent's communication is achieved through asynchronous and message-oriented interactions. In addition to physical connection, it requires semantics in order to enable agents to reach the concluding state of the interaction through common understanding of the messages and their meanings. Consequently, the dialogue ontology and conversational semantics has to be defined within the framework of a conversation space. This space is defined as a sequence of messages exchanged between agents following a (set of) defined dialogue protocol(s). Dialogue protocols enable agents to take part in conversations by committing to the shared protocol semantics and defined conversation space within which agents are enabled to act, still preserving their decision-making autonomy (Greaves et al., 2000). The space boundary is defined by the definition of (1) an ontology of common terms that agents need to agree upon, (2) a language for ontology description, and (3) a language for a conversation description.

There are several conversation models identified in existing literature. A Belief-Desire-Intention (BDI) scheme is described in (Bratman et al., 1988) but it suffers from the semantic verification problem (Wooldridge, 2000). Although the MAP (Walton, 2002) language introduces an interesting concept of scenes and roles, its agent-centric nature increases complexity of conversation verification where inconsistent and unstructured protocols are not easy to detect. (Endriss et al., 2003) introduces the idea of protocol definition within the agent's own business logic. The similar idea is present in the IOM/T (Doi et al., 2005) language where the main focus is on message flow between agents. On the other hand, the language relies on a particular agent runtime platform which prevents it to be considered for large-scale adoption.

In this paper, we introduce a SDLMAS language for scenario description in multi-agent systems. The language is independent of the agent runtime platform and implementation language, and it is focused on a definition of message flow between agents providing intuitive scenario description. The paper also addresses corresponding SDLMAS platform purposely built for rapid design and development of agent-based systems. The platform includes a code generator from the SDLMAS language into a target implementation language and an agent platform. The generated code integrates seamlessly with the rest of the platform that provides scenario execution runtime framework, leaving complexity of interactions and message propagation hidden from the developer. Obviously, the agent's own business logic which is domain specific, provided as a generated code skeleton, needs to be populated manually with the desired functionality.

The SDLMAS language was developed in support of multi-agent simulation system for prediction and scheduling of engine overhaul in aerospace industry (Stranjak et al., 2008). Complexity of scenario descriptions between engine fleet managers and engine repair stations required significant design, development and testing efforts. Such difficult problem would require a declarative language for conversation framework definition within which simultaneous interactions would occur without explicit specification of their ordering or timing. In addition to this, a requirement to define intertwining protocols was necessary to enable cross-scenario communication. In other words, this would allow agents to achieve given tasks or gather required information within one scenario and to communicate them to the other. These agents would need to maintain several conversations simultaneously during negotiations for the best shop visit time and repair slot while utilizing balance between revenue earned from the engines in service and an acceptable risk of disruption. Current scenario description languages cannot satisfy given requirements and therefore it was necessary to define a new interaction description language and to build a corresponding platform.

The SDLMAS platform was developed as a development and runtime environment for the multi-agent systems whose scenarios are described using the SDLMAS language. The platform equips a designer with the mechanisms to define negotiation scenarios and a developer with the automatically generated components for message passing, execution of scenario actions and invocation of given business procedures, allowing him or her to concentrate only on development of domain specific actions that agent needs to perform during conversations. This way, the development cycle was shorten several times with significantly increased reliability of the system after deployment.

2. Related work

As a part of an initiative to formalize and define the process of design and implementation of multi-agent systems, numerous models and methodologies (Gomez-Sanz et al., 2003) (Wood et al., 2000) (Wooldridge et al., 2000) were developed in the last several decades. Agent interactions are ingredient element of design, development and deployment of such systems.

Agent UML is one of the popular models to visually represent agent interactions (Paurobally et al., 2003a). It is based on the UML standard in order to mitigate a paradigm shift from object-oriented to agent-based concepts, and to enable standardized notation for analysis, design and implementation of agent systems. In order to include concepts of roles and behaviors, UML class and sequence diagrams are extended by specificities of agent interactions. Since AUML is a visual language, it is necessary to define the ways of its transformation into textual notation of protocols in order to achieve practical usability, and various language transitions are proposed (Warmer et al., 1999) (Winikoff, 2005). In spite of these efforts, AUML lacks enough representation capabilities of agent's states, which causes an inability to define conditions under which messages can be received or sent by an agent. Additionally, it lacks the expressiveness and clarity, especially in case of complex multi-lateral scenario definitions. Although AUML lacks some features mentioned above, it has influenced other language designs (Warmer et al., 1999) (Winikoff, 2005) (Dinkloh et al., 2005) (Huget, 2002) and modeling frameworks (Quenum et al., 2006). The OCL language (Warmer et al., 1999) is the way to represent UML in a textual format and it can be used to represent AUML too but it was shown to have limited usability (Richters et al., 1998).

Petri Nets (Cost et al., 1999) also offers interesting perspective on agent-based scenario definitions, but due to lack of clarity and scalability for medium-to-complex scenarios, it is not very appropriate for description of agent conversations, especially for popular protocol like Contract-Net (Purvis et al., 2002).

Belief-Desire-Intention (BDI) scheme (Bratman et al., 1988) is another popular model for description of agent's functionality through definition of agent's goals that should be fulfilled by executing tasks based on knowledge base about its immediate environment. Although the model influenced definitions of widely accepted FIPA and ACL (FIPA) standards, it suffers from the "semantic verification" problem (Wooldridge, 2000) where it is not possible to guarantee the equal understanding of ontology terms by all agents involved.

The MAP language (Walton, 2003) for dialogue protocol definition is based on the principles found in Electronic Institutions (Estava et al., 2001) while its semantics is inspired by logic which is basis for communication and parallel systems (Milner, 1989). The language organizes dialogue definitions in scenes, an interaction context within which agents are communicating with each other. Another key concept is an agent's role, a certain set of behaviors that an agent will adopt during interactions. An interaction protocol is adopted by an agent based on the role agent plays. The MAP language is not based on message flows as the way how protocol is defined but on agent-centric approach. This means that agent's action is defined separately for every agent which implies significant effort of protocol validation and analysis since the protocol definition is scattered across several agents.

The IOM/T language (Doi et al., 2005) introduces a formal way of mapping interaction protocol from AUML and in general emerged from a tendency to strictly define protocols in a textual notation. Unlike the MAP language, the language is based on message flow which means the definitions of protocol-related agent activities are not separated but defined in the

single definition. Unfortunately it lacks some important characteristics related to scenario descriptions.
Firstly, there is no explicit definition of message performative used in conversations. A protocol is defined as a sequence of messages whose character is determined by corresponding performatives. If the language lacks formal explicit definition of message performatives, message validity needs to be performed within the agent's business logic. This increases the complexity of the logic implementation with additional performance penalties since irrelevant or protocol-incompatible messages will not be immediately rejected after their arrival but during their processing. Furthermore, it is not possible to determine the differences between conversational and terminating performatives which prevents an agent to explicitly recognize a conversation completion. Since there is no clear distinction between performative types, it is also not possible to define conditions under which messages can be forwarded to the business logic or just ignored.
Secondly, the cardinality of agent instances is bound to the protocol implementation inside agent's internal logic. The protocol description is linked with the given scenario and the number of agent instances in a dialogue. If it is required to change a cardinality of agent instances, the protocol description needs to be modified too in order to reflect this update since the business logic defines the interaction description including agent instance cardinality. Consequently, it is not possible to change number of instances of a particular agent without re-implementing the internal implementation.
Thirdly, an implementation of agent's internal logic is language-dependent on the JADE agent platform (JADE), which disables possibility of usage IOM/T language in another agent platform.
Q Language (Ishida, 2002) is another language for interaction protocol definition in a textual form. Its main purpose is to define interactions with a user or other agents on behalf of a user. Consequently, it is not based on message flows in the way that SDLMAS is. Typically, scenario description will be considered only from the point of view of one agent who is supposed to interact with other parties assuming their prior knowledge about the scenario they are supposed to follow.
Popularity of scenario-based development of multi-agent systems is increasing and several interesting applications can be found in rescue simulation (Shinoda et al., 2003), interactive TV program (Shirai et al., 2007), modeling and simulations (Donikian, 2001), etc.

3. Scenario description language

The SDLMAS scenario description language, together with the SDLMAS run-time framework, represents the core component of the SDLMAS platform. The language has been designed with the main purpose of rapid design and development of multi-agent systems. In order to fulfill those goals three main language properties had to be achieved:

- Simple and intuitive creation of scenarios in multi-agent systems but yet capable enough to support complex agent interaction descriptions,
- Expressive language for strict and flexible interaction within the described multi-agent system,
- Abstraction of run-time environment in order to protect its users from run-time complexities related to interaction management within a large multi-agent system.

The SDLMAS language is a declarative, interaction-centric description language, designed with the purpose of defining permitted sequences of messages (communicative acts) exchanged among interaction participants (agents in a multi-agent system). An interaction protocol is defined implicitly, as a sequence of agent actions where order of those actions is important, thus achieving a sequential approach in protocol definition. The language describes conversations among agents as a sequence of conversation actions, where actions define a conditional mapping between incoming and outgoing messages, and an agent's internal logic. Conditions for reception and transmission of messages are defined explicitly as a part of a conversation protocol definition. An elementary action of the language is defined as a procedure that will be executed as a consequence of a condition being satisfied following reception of one or more messages. A scenario is formed of a logically complete sequence of conversation actions aimed at achieving some rational effect in the multi-agent system.

The language is restrained to a communication aspect of multi-agent systems, thus enforcing strict separation between conversation actions and internal agent logic implementing agent reasoning. Although direct influence on agent decision process is not possible, inadequate expressiveness could indirectly restrict or bias the way agent reasons or acts, or simply fail the attempt to model interactions of required complexity. The SDLMAS language provides adequate expressiveness through achievement of the following:

- Parallelism in agent interactions, defining synchronization communication points,
- Definition of conditions on incoming and outgoing messages, including complex logical expressions based around message types and agents as sources or targets of incoming and outgoing messages,
- Controlled variations in message exchanges during scenario execution, effectively defining a set of allowed scenario instances from one scenario definition,
- Complex runtime interaction within non-trivial multi-agent systems are mainly caused by following requirements:
    - To support many scenario instances an agent can concurrently participate in,
    - To handle simultaneous conversations with several agents within potentially several scenario instances of the same scenario, and to ensure conversation adherence to scenario defined rules,
    - To maintain conversation state with a particular agent within a scenario instance and correctly terminate conversations,
    - To correctly handle exceptions during conversation, both at the execution and at the protocol level.

Most of these required system properties are hidden from the developer by the built-in platform run-time mechanisms. Developed scenarios are completely independent of the agent platform and implementation language due to language's declarative nature and strict separation of communication aspects of multi-agent system from implementation of agents' internal business logic. As such, they allow relatively simple analysis and consistency validation, as well as transformation into alternative models. As the final step in the process of modeling and designing interactions within a multi-agent system, defined interaction scenarios are transformed into a program code for target agent platform and implementation language.

3.1 Example scenarios

In this chapter two example agent interaction scenarios are given in order to present the key elements and characteristics of the SDLMAS language. The examples are focused on a multi-agent system concerned with electrical power consumption planning and run-time power reallocation. All the power consumers, producers and brokering system elements are represented as one or more intelligent agents forming a virtual energy market. The main focus of the system is to ensure short and long term power allocation among consumers and power usage patterns such that the goals of the system can be fulfilled as optimally as possible, depending on variable internal and external system conditions. The three presented scenarios are just a small subset of scenario definitions required to completely define a conversational behavior of the system.

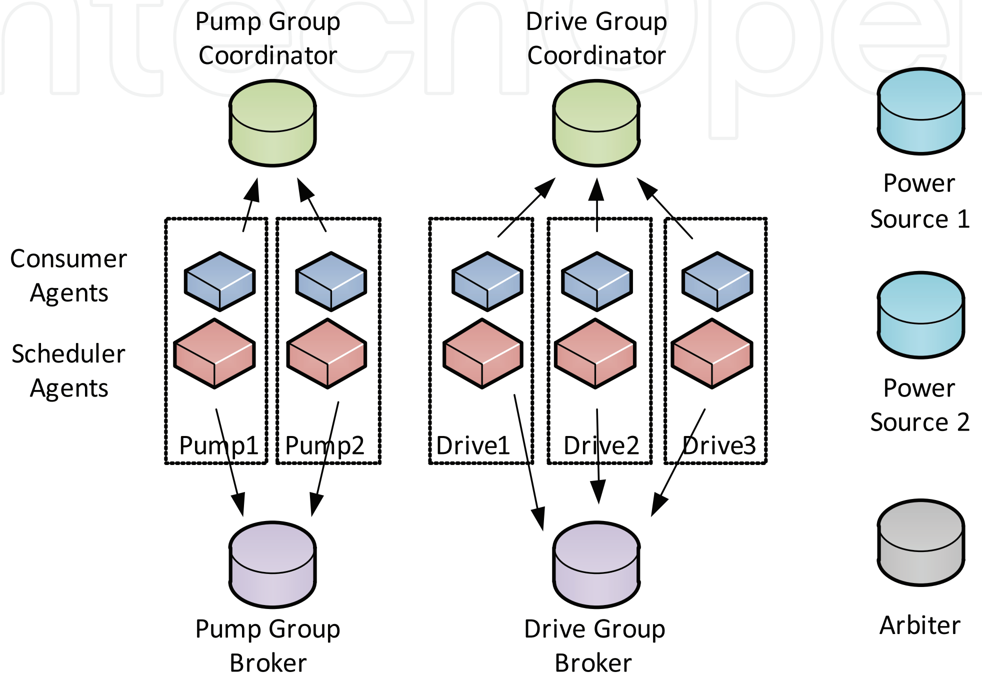

Fig. 1. Example multi-agent system

The structure of the described multi-agent system is depicted on Figure 1 including all agent types, their relations and mapping of agent types to physical system components. Each physical power consumer within the described system is represented by one Consumer type agent, concerned with current and near-future power consumption, and one Scheduler type agent in charge of power consumption planning and power allocation. Power consumers are organized into consumer groups of similar characteristics and managed by Coordinator agents. Pre-allocated power reservations from various system power sources are held by Coordinator agents allowing them to immediately serve consumer demands for additional power, if existing allocations suffice. If not, additional power is sought by coordinators from two agent types: Broker agent types and Power Source agent types. Such a process requires complex and simultaneous negotiations among many power sources and coordinator agents in order to identify the best offer or a combination of offers that would meet the initial request at an acceptable price. Broker agents negotiate power requests originated from Coordinator agents, examine and coordinate power reservation changes with Scheduler agents within their coordination group, and sell power surpluses in exchange for future power reservations or an immediately collected fee. Power Source agents represent various power sources in the system and their inherent characteristics (availability, current price etc).
The DynamicPowerRequest scenario depicts a situation where a consumer, a high-pressure pump, is required to complete an unplanned action and requires additional power to

perform it. Such power reallocation must be negotiated in a specified time frame and with minimal impact on other system functions.

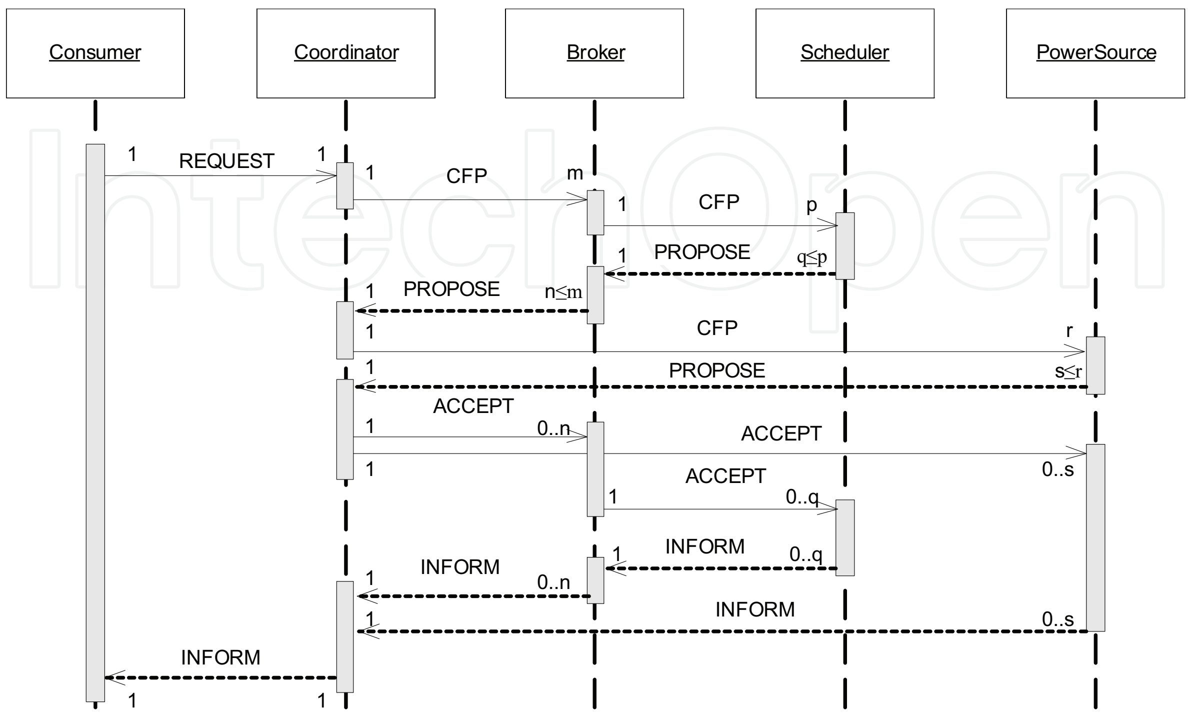

Fig. 2. Sequence diagram of the DynamicPowerRequest interaction scenario

Request for additional power is delivered to the agent's group Coordinator agent, where a decision is made based on a locally available pre-allocated power surplus. A requested amount of power can immediately be allocated to the pump if the sufficient amount of power is available to the coordinator, or the power can be sought from other sources such as other consumer groups and active power sources. In order to obtain additional power a CFP is issued by the coordinator agent to all the other coordinators in the system (represented by Power Broker agents). Power Broker agents can immediately deny or propose release of their pre-allocated power surpluses for compensation, or they can consult their managed Scheduler agents, by issuing a CFP, for any excess power or plan rescheduling. In any case, an offer or a refuse message is returned to the requesting Coordinator agent, determined on the basis of availability of pre-allocated Power Broker power or willingness of consumers within a Power Broker coordinated groups to release (sell) their power reservations. A call for proposal is also issued by the Coordinator agent to all of the system's active power sources and power production proposals collected. All the received offers are evaluated on the basis of their temporal characteristics and compensation requested from the originating source (both in form of future power allocations and immediate fees), and the most affordable one is selected.

The result of this complex interaction among the system's actors is finally relayed to the requesting high-pressure pump power consumer. If no satisfactory offer has been received, the requested unplanned action cannot be performed. Figure 2 contains a very simplified sequence diagram of the described interactions, lacking most of the details concerning alternative conversation paths and hiding a large amount of possible variations within the execution of scenario. The key characteristic of the presented interaction that makes

(A)UML sequence diagrams less suitable is its intertwining description of three contract net protocols where results from one protocol influence execution of other protocols. In addition, there are large sections of parallel conversations among agents and optional executions of certain scenario parts, that makes rendering of their description in UML very complicated and hard to interpret.
The *PowerAuction* scenario describes a simplified version of the regular power allocation mechanism to all interested Broker agents (details of interactions between Broker and Scheduler agents have been omitted for clarity reasons). A Power Source agent with available power within a certain time frame announces power availability to all Broker agents within the system. Such an announcement must be approved by the Arbiter agent (only one instance of Arbiter agent is present in the system). In the first step, Broker agents express their interest by submitting the first bid or leave the auction scenario immediately. As the next step a variant of English auction is employed to allocate available power to one or more highest bidders. At the end of auction participating Broker agents are informed of the final auction result by the Power Source agent. A simplified UML diagram of the interactions is presented in Figure 3.

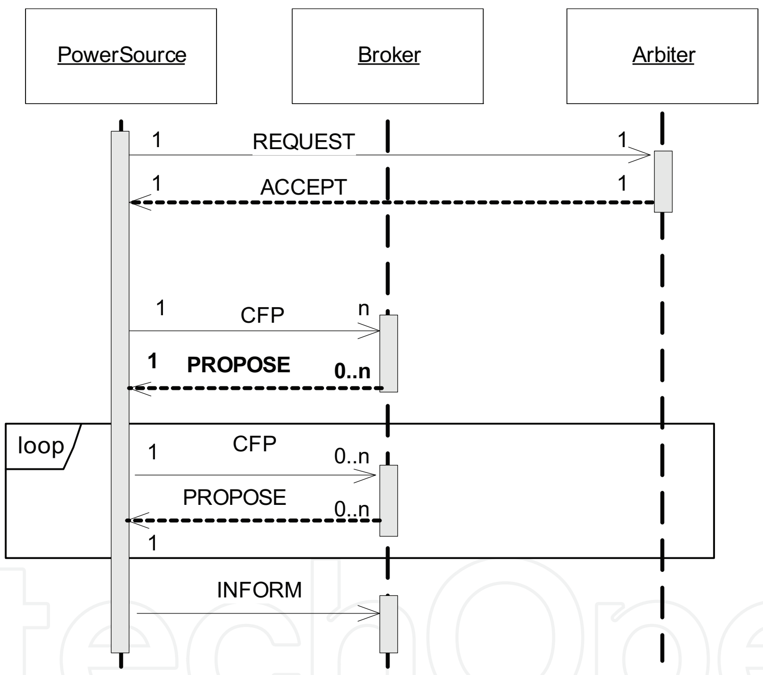

Fig. 3. Sequence diagram of the Power Auction interaction scenario

3.2 SDLMAS Scenario definition example

```
agent @consumer       : Consumer
agent $coordinator    : Coordinator;
agent @coordinator    : Broker;
agent $consumer, $localConsumers : Scheduler;
agent @psource        : PowerSource;
agent arbiter         : Arbiter;

scenario DynamicPowerRequest [reentrant=no] {
```

```

action Consumer in powerRequest() {
  msgSnd : (REQUEST -> $coordinator);
}

action Coordinator in rcvPowerRequest() {
  msgRcv : (REQUEST <- @consumer);
  msgSnd : (CFP -> <Broker>) |
           (#INFORM_DONE -> @consumer);
}

action Broker in rcvPowerReleaseCFP() {
  msgRcv : (CFP <- @coordinator);
  msgSnd : (CFP -> <$localConsumers>) |
           (PROPOSE | #REFUSE -> @coordinator);
}

action Scheduler in rcvPowerReleaseCFP() {
  msgRcv : (CFP <- @coordinator);
  msgSnd : (PROPOSE | #REFUSE -> @coordinator);
}

action Broker in collectPwrReleaseRsp(){
  msgRcv : (PROPOSE | #REFUSE <- <$localConsumers>);
  msgSnd : (PROPOSE | #REFUSE -> @coordinator);
}

action Coordinator in collectPwrReleaseRsp() {
  msgRcv : (PROPOSE | #REFUSE <- <Broker>);
  msgSnd : (CFP -> <PowerSource>);
}

action PowerSource in processPwrRequestProposal() {
  msgRcv : (CFP <- @coordinator);
  msgSnd : (PROPOSE | #REFUSE -> @coordinator);
}

action Coordinator in decideOnPwrAllocationOffers() {
  msgRcv : (PROPOSE | #REFUSE <- <PowerSource>);
  msgSnd : (ACCEPT | #REJECT -> <Broker>) &
           (ACCEPT | #REJECT -> <PowerSource>);
}

action PowerSource in allocatePower() {
  msgRcv : (ACCEPT | #REJECT <- @coordinator);
  msgSnd : (#FAILURE | #INFORM_DONE -> @coordinator);
}

action Broker in releasePower() {
  msgRcv : (ACCEPT | #REJECT <- @coordinator);
  msgSnd : (ACCEPT | #REJECT -> <$localConsumers>);
```

```
 }

 action Scheduler in confirmPwrRelease() {
   msgRcv : (ACCEPT | #REJECT <- @coordinator);
   msgSnd : (#FAILURE | #INFORM_DONE -> @coordinator);
 }

 action Broker in collectConsumerCommits() {
   msgRcv : (#FAILURE|#INFORM_DONE <- <$localConsumers>);
   msgSnd : (#FAILURE | #INFORM_DONE -> @coordinator);
 }

 action Coordinator in collectPwrAllocationCommits() {
   msgRcv : (#FAILURE|#INFORM_DONE <- <Broker>) &
            (#FAILURE |#INFORM_DONE <- <PowerSource>);
   msgSnd : (#REFUSE | #INFORM_DONE -> @consumer);
 }

 action Consumer in receivePwrAllocationRsp() [timeout=2000] {
   msgRcv : (#REFUSE | #INFORM_DONE <- $coordinator);
 }
}

scenario PowerAuction {

 action PowerSource in requestAuction() {
   msgSnd : (REQUEST -> arbiter);
 }

 action Arbiter in auctionRequest() {
   msgRcv : (REQUEST <- @psource);
   msgSnd : (ACCEPT | #REJECT -> @psource);
 }

 action PowerSource in initialCFP() {
   msgRcv : (ACCEPT | #REJECT <- arbiter);
   msgSnd : (CFP -> <Broker>);
 }

 action Broker in auctionStart() {
   msgRcv : (CFP <- @psource);
   msgSnd : (PROPOSE | #REFUSE -> @psource);
 }

 loop {
   action PowerSource in cfp1() {
     msgRcv : (PROPOSE | #REFUSE <- <Broker>);
     msgSnd : (CFP -> <Broker>);
   }

   action Broker in loopProps() {
```

```
111      msgRcv : (CFP <- @psource);
112      msgSnd : (PROPOSE | !REJECT -> @psource);
113    }
114
115    action PowerSource in cfp2() {
116      msgRcv : (PROPOSE | !REJECT <- <Broker>);
117      msgSnd : (#INFORM -> <Broker>);
118    }
119  }
120
121  action Broker in endAuction() {
122    msgRcv : (#INFORM <- @psource);
123  }
124 }
```

3.3 SDLMAS elements

SDLMAS language describes conversational behavior of multi-agent systems based on a number of agent types (roles), agent references and a set of interaction scenarios. Each scenario involves all or a subset of defined agent references in a sequence of conversational actions performed by involved agents. SDLMAS scenario descriptions are stored in a form of a text file where the header part contains the definitions of agent types and agent references, while the rest of the file (body) contains one or more scenario definitions.

3.3.1 Roles, scenarios and conversation actions

SDLMAS defines roles as standardized patterns of behavior required of all agents participating in conversations. SDLMAS employs an implicit approach to role behavior definition, as opposed to commonly used explicit state-based agent-centric approach. Each agent role behavior is defined by a number of role belonging conversational actions within all scenarios.

Inter-agent communication is not addressed on the single agent level, but defines communication patterns among different agent roles, thus leaving concrete scenario execution to adapt to current system structure. A single agent within a concrete multi-agent system can be assigned more than one role, and can participate in many parallel scenario instances. A metadata mechanism can be used to control specific execution behavior of scenarios, roles or actions in such a case.

At line 5 of the example the Power Source role is declared together with an anonymous reference of the same type. Power Source role behavior is defined by a sequence of conversation actions processPwrRequestProposal and allocatePower from DynamicPowerRequest scenario and requestAuction, initialCFP, cfp1 and cfp2 conversation actions from PowerAuction scenario.

SDLMAS scenario is defined by a unique scenario name and a sequence of conversation actions implicitly defining one interaction protocol. In a given SDLMAS example, scenario declarations start at lines 8 and 83.

Conversation actions are defined within scenario scope and are attached to agent roles. Each conversation action defines a procedure and two communication conditions. The procedure represents an internal agent logic function, and is invoked by the SDLMAS runtime framework upon satisfaction of communicative preconditions. Communicative postconditions ensure that all messages generated by internal agent's procedure conform to message transmission conditions. Regular conversation actions are defined using both preconditions and postconditions, but other types can have incomplete conversation conditions: scenario triggering action (first scenario action) does not define a precondition and conversation terminating action (last scenario action) does not define a postcondition.

Communicative preconditions and postconditions are defined with respect to message performative(s) and message originating agent role(s). A communicative precondition defines circumstances under which a received message or a set of received messages are being passed to an internal procedure implementing agent logic. A condition consists of a list of expected message performatives and their originating agent roles, and can form expressions using logical operators. Communicative postcondition lists messages and their performatives, generated by an execution of an internal agent logic procedure, to be sent to corresponding agent roles.

A simple communicative action condition is formed of an atomic communicative condition consisting of a required message performative and a target or source agent reference (examples on lines 38 or 86). A more flexible atomic condition definition is allowed by stating a list of required performatives separated by *or* operator symbol '|', defining "one of" semantics (line 28). Complex conditions are formed of a number of atomic conditions combined using logical operators *or* and *and* (symbol '&') in conjunction with parenthesis as grouping operator (line 22).

Sequential nature of conversations among agents can be described using only simple conditions. In the example scenario PowerAuction, a Power Source agent first requests a clearance from arbiter agent (line 86), and only after the clearance is granted (ACCEPT performative received) call for proposal messages are sent to all Broker agents in the system (line 96). Achieving conversation parallelism among agents of differing roles requires usage of complex conversation conditions. For example, Coordinator agent accepts or rejects Broker and PowerSource proposals in parallel (lines 48 and 49), effectively defining a "span" point (decideOnPwrAllocationOffers) and a corresponding "join" point in action collectPwrAllocationCommits (lines 73 and 74).

3.3.2 Conversation context

A conversation context represents a scenario instance execution within an agent. It encapsulates all the elements that define the scenario instance state, such as state of the conversation (current conversation action), communicative conditions and active constraints, values of agent references etc. A new conversation context is created as a consequence of two different communication events. The first event is the activation of the scenario triggering action, regardless of the activation method used (external or internal). Created conversation context is called a root conversation context and is a parent context to all the conversation contexts created during the scenario instance execution. In this case the initiator and the owner of the context is the agent that triggered the scenario. A conversation context is also created when an agent receives a message that activates the first conversation action in a scenario for a role the agent plays. The initiator of this context is the agent that initiated the conversation (triggered the context creation). A parent-child context relation is created in this case, forming the tree structure of conversation contexts with root context as the owner. Termination of a parent context implies termination of all contexts in the parent context sub tree. On Figure 4 a complete conversation context tree is presented for a

DynamicPowerRequest scenario execution, where pump1 Consumer agent requests additional power.

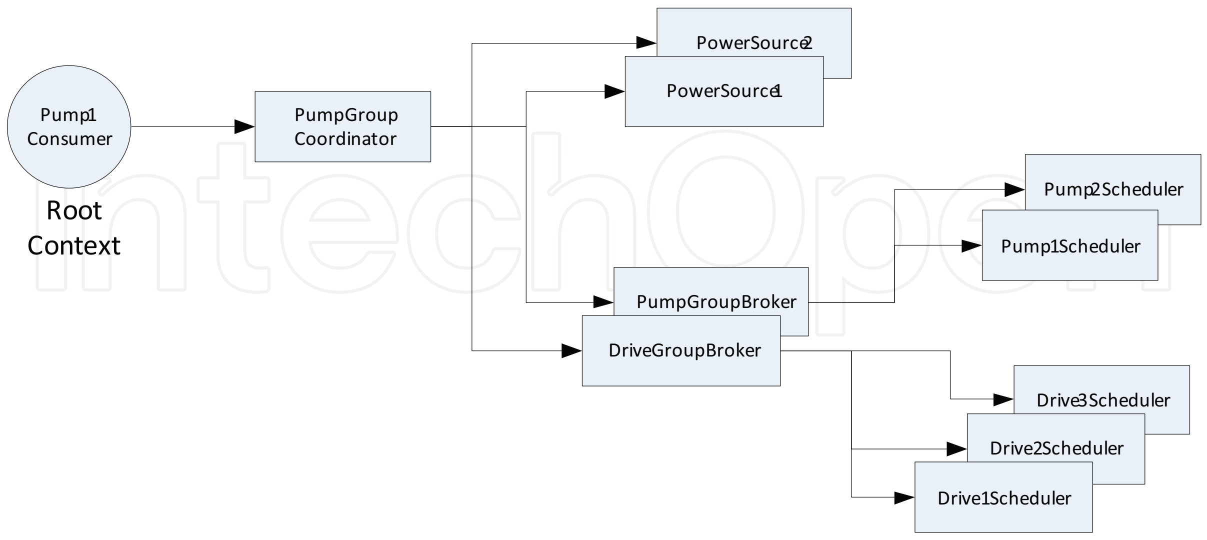

Fig. 4. Conversation context tree for Dynamic Power Request scenario execution

A single agent can be simultaneously involved in more than one scenario execution, thus having more than one active conversation context. In case an agent is assigned more than one role, and those roles participate in the same scenario, a situation can occur where the same conversation context is recursively activated (reentrant context). Reentrancy can be disabled on the scenario level by explicitly setting scenario property using metadata mechanism (line 8).

3.3.3 Conversation loops

In SDLMAS, a sequence of recurring conversational actions is defined as a loop. In scenario definition, recurring actions are isolated within a loop scope, defined by a loop keyword and curly braces (lines 104-123). Loops are a special version of nested scenarios, not addressed in this text. Additional requirements and semantics are defined for loop scoped conversation actions:

- The first and the last conversation action must be attached to the same agent role,
- Loop must be terminated by loop context termination (exchange of loop terminating performatives) or scenario context termination (exchange of scenario terminating performatives),
- Communicative precondition of the first conversation action and the postcondition of the last action are not part of the loop,
- The first action precondition is evaluated only once, upon start of the loop scenario execution. At line 106, PROPOSE or #REFUSE performatives are received from Broker agents only once, and CFP sent (line 107) to those who had not left the scenario with terminating performative #REFUSE,
- The last action postcondition is evaluated only once, upon loop termination by exchange of loop terminating performatives (lines 112 and 116), when all Broker agents refuse to submit a new bid,
- On all but first loop cycle, communicative precondition of the last action is effectively treated as the precondition of the first action (i.e. line 116 is ‘copied’ to line 106).

A loop conversation context is created upon when scenario execution enters the loop. All reference values are copied to the loop conversation context. Depending on the performative type used, group membership modifications are either confined to loop scope (loop terminating performatives) or propagated to scenario conversation context (scenario terminating performatives). When scenario execution leaves the loop, loop conversation context is collapsed, and original scenario context is again declared the active one.

3.3.4 Agent references

Agent references represent a single agent or a group of agents of a specified role. All agent references must be declared in the head section of SDLMAS file. A reference is characterized by its role, cardinality (single or group) and binding method. Five agent reference types are used in the current version of the SDLMAS language: *verbatim*, *variable*, *anonymous, group* and *group variable* references. Multiple conversation contexts active on the same agent have distinct reference bindings. All but verbatim references retain their bindings only for the duration of their enclosing conversation context.

Verbatim reference value is fixed at scenario definition and represents a specific agent within the system (usually references a singleton role agent). The example system hosts only one agent of role Arbiter, whose verbatim reference 'arbiter' is defined on line 6. Line 86 contains usage of that reference, where REQUEST performative is sent by a PowerSource role agent to the 'arbiter' agent.

Variable reference value must be bound by the agent logic during a new conversation context initialization and retains its initial value throughout the context lifetime. Example declarations of variable references (reference name prefixed by '$' symbol) can be found on lines 2 and 4. In powerRequest action (line 11) an agent of type Consumer initiates a DynamicPowerRequest scenario by sending a REQUEST message to its coordinator agent (each Consumer agent must be aware of its Coordinator agent).

Anonymous reference (names prefixed with '@' symbol) value binding is performed by the framework during creation of a new child conversation context - triggered by the reception of a scenario-triggering message. Lines 1, 3 and 5 contain declarations of anonymous references. Line 27 contains an example of an anonymous reference binding, where a new child conversation context is created in one of Scheduler agents and reference @coordinator is assigned to agent of type Broker, the one who dispatched a CFP message to the particular Scheduler agent. Anonymous reference value is preserved throughout the conversation context lifetime. Lines 63 and 64 contain an example of anonymous reference usage during the conversation context lifetime. As the reference value is invariable, reception and transmission of messages with @coordinator agent is performed with the same agent who triggered the conversation context creation at line 27.

Group references refer to more than one agent at a time. There are two types of group references defined in the current version of SDLMAS; *role group* references and *variable group* references. Both group references are populated at the time of conversation context creation. During conversation context lifetime agents can retain or leave the membership of a group reference, but new agents cannot be added to once initialized group reference. Agents leave a membership as a consequence of explicit exchange of scenario- or loop-terminating performatives, or an implicit reception of #TIMEOUT performative.

Role group reference name is enclosed within angled brackets and denotes all agents of a particular type (role) present in the system at the moment a group population is bound to a reference. Population identification and binding is performed by the SDLMAS platform during runtime and is hidden from scenario designers and developers. Line 16 of the DynamicPowerRequest scenario contains an example where a Coordinator agent dispatches a CFP message to all of the Broker agents, and at line 37 collects responses. Broker agents that terminated a conversation by returning a #REFUSE message are removed from the group reference membership.

Variable group reference name is prefixed with '$' symbol and enclosed within angled brackets. Those references differ from role group references only in the method of agent biding; while role group references are populated externally (by the SDLMAS runtime), group variable references are populated by internal agent logic, and follow the same philosophy as variable references. Group variable reference usage example can be found at line 59, where a Broker agent sends an ACCEPT or #REJECT message to all the Scheduler agents currently bound to the group variable.

3.3.5 Performatives

SDLMAS differentiates among four performative types: *conversational*, *scenario-terminating*, *loop-terminating* and *implicit performatives*. Conversational performatives preserve the active conversation context. All performatives not prefixed with special symbols '!' and '#' are conversational performatives.

Scenario-terminating performatives, prefixed with '#' symbol, explicitly denote an end of conversation between two or more agents within an active scenario instance. Run-time effect of scenario-terminating performative is determined by the hierarchical relation between affected conversation contexts: child context is being terminated and parent context performs revision of group reference memberships and, if, necessary, constraint elimination process. If the result of the constraint elimination process is an empty constraint structure, parent scenario instance is also being terminated. Termination of root context implies a scenario instance termination. An example of scenario-terminating performative usage can be found on line 43, where PowerSource role agent communicates a PROPOSE (conversational) or #REFUSE (scenario-terminating) performative to the requesting Coordinator role agent.

Loop-terminating performatives, prefixed with '!' symbol, explicitly denote an end of loop scoped conversation between two or more system agents. In case of loop terminating performative being communicated, child loop conversation context is being collapsed and parent loop context undergoes a procedure similar to one in case of scenario-terminating performative. The major difference between scenario-terminating and loop-terminating performatives is that termination (collapse) of loop contexts does not affect states of base conversation context references and constraints (their values are restored after the loop scoped interaction is terminated), but scenario-terminating performatives do. If the PowerSource agent (line 106) receives a #REFUSE performative from one of the Broker agents, that agent is evicted from the <Broker> group reference until the end of scenario execution. But if PowerSource receives a !REJECT performative from another Broker agent (line 116), that Broker agent is temporarily evicted from the <Broker> group reference only until the end of active loop. Consequently, #INFORM performative (line 117) will be sent to all Broker agents that had not communicated a #REFUSE performative or whose response had been timed out and replaced with implicit #TIMEOUT performative.

Implicit performatives are performatives generated by the runtime SDLMAS system in response to a specific event. Implicit performatives can be conversational (explicitly defined using scenario metadata mechanisms) or scenario-terminating. #TIMEOUT performative is generated by the SDLMAS runtime every time a message from a certain agent is not received within an expected time frame. On line 79, if #REFUSE or #INFORM_DONE performative is not received within 2 seconds from sending the REQUEST performative to the $coordinator agent (line 11), the Consumer agent will receive a #TIMEOUT performative instead.

3.3.6 Metadata

In order to provide a system designer with a mechanism to express required runtime scenario execution properties, a concept of scenario metadata definition has been introduced in the SDLMAS language. Metadata is expressed in form of a sequence of semicolon separated name-value pairs enclosed in square brackets and are placed immediately beside declarations of affected scenario elements. Line 78 of the example contains a timeout metadata definition, effectively forcing a #TIMEOUT performative to be generated if no response has been received from $coordinator agent within 2 seconds after the REQUEST performative had been issued (line 11). Metadata properties are defined on a per-language element basis (role, scenario, action).

4. SDLMAS platform

SDLMAS platform provides tools and a framework for implementation and runtime support of multi-agent systems whose interactions are modeled using SDLMAS language.

4.1 SDLMAS platform components

The following components make a set of core elements of a multi-agent system that is developed using the SDLMAS platform: a generic SDLMAS component (Management agent), application-specific SDLMAS components (Application agents) and underlying agent platform components (ORB and Naming Service agent).

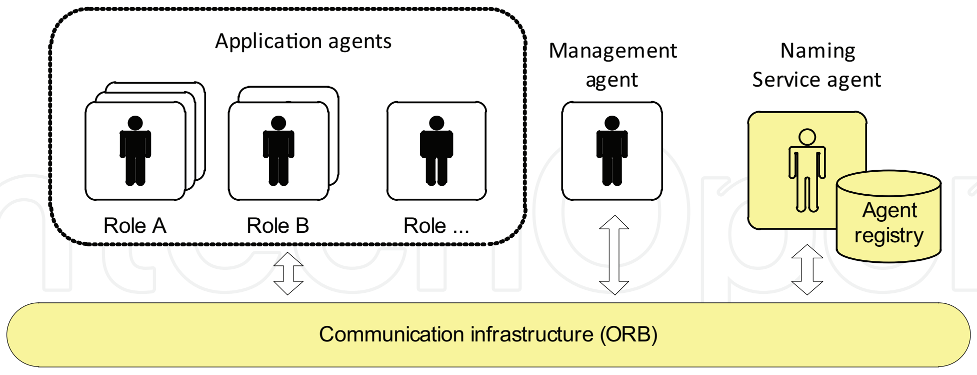

Fig. 5. SDLMAS Platform

The SDLMAS platform (Cavrak et al., 2009) relies on core functionality provided by target agent platform such as agent creation and multithreaded execution of agents, FIPA compliant messaging, agent container management, naming service as a central storage of agent references, etc. Upon their successful initialization, all agents are required to register with the Naming Service agent, and to deregister prior to their deactivation. Accurate information stored in the registry is crucial for correct behavior of late agent reference binding mechanism:

- Agent's type (role) is verified using the information from the registry in order to bind an anonymous reference to a real agent reference,
- All agents of the required type (role) are collected from the registry in order to bind a group reference to a list of acquired agents references.

Management agent, a mandatory system element, provides a support for bootstrapping and initialization of a multi-agent system based on provided global and agent-specific configurations.

Functionality of a multi-agent system is based on individual functionalities of system-constituting entities and on effects of interaction among those entities. A SDLMAS application agent plays a particular role in the system as it is defined in the scenario description. Conformance to the given role is guaranteed by generated program code from the description and must not be modified by an agent developer. Other portions of agent code, related to internal agent logic, are partially generated code skeletons where a developer is required to only implement agent's procedures within already predefined procedure signatures and parameter definitions.

System start-up procedure is as follows: (1) Naming Service Agent and Management Agent are started, (2) based on system configuration and scenario descriptions, the Management Agent starts a number of Application Agents, (3) Application Agents register at the Naming Service Agent, (4) a number of scenarios are initiated by the Management Agent based on scenario descriptions, with specified application agents as their initiators.

4.2 Automatic code generation

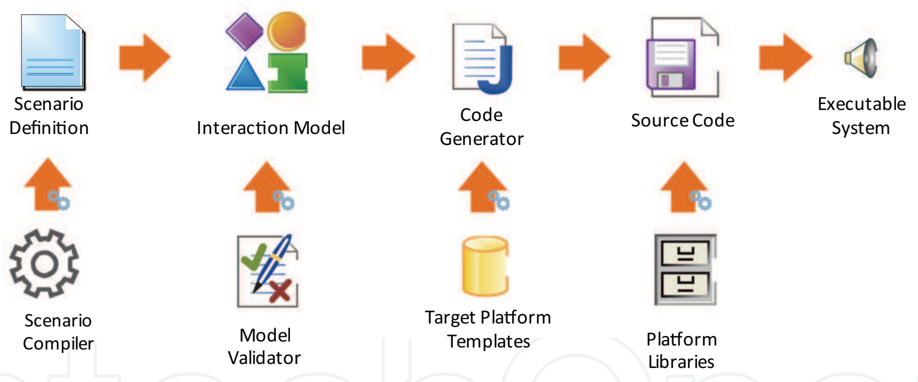

Fig. 6. SDLMAS Code Generation

The process of converting a SDLMAS scenario definition to a source code for a target agent platform is depicted on Figure 6. A text file containing agent type declarations and scenario definitions is converted to an internal model, suitable for both scenario validation and code generation. Platform-specific set of code templates are used to generate source code for the target agent platform. This way, the flexibility and transparency in choosing another target agent platform or implementation language is achieved as this approach allows easy retargeting of generated agent code by using different code templates. Generated source entities are divided into two main categories: model-level entities, shared among many system components, and scenario-level entities, containing role- and scenario-specific implementation of agent communication behavior.

SDLMAS scenario definition can be converted into platform code in two ways: a command line based tool and an Eclipse plug-in. The plug-in allows for easier syntax checking, model

transformation and validation, as well as navigation between scenario definition and generated implementation code. Internal interaction model is stored in an EMF-based model and JET templates are used for generating the Java code. Currently the JADE platform (JADE) is supported.
A SDLMAS agent implementation can be divided into three main layers: Platform Abstraction Layer (PAL), SDLMAS Platform Layer (SPL) and Application Logic Layer (ALL).
PAL abstracts the specificities of the target platform programming interfaces and semantics and provides the SDLMAS platform with the generic interface towards the target platform resources. This layer is clearly dependent on the target platform and it must be developed for each platform the SDLMAS is ported to.
SPL consists of two sets of components: generic components and scenario-dependent components. Generic components provide support for scenario- and role-independent functionality and is provided in a form of a library. Scenario-dependent components are automatically generated from SDLMAS scenario description and it contains necessary functionality for tracking conversation progress and its contexts, enforcing message reception and transmission conditions and invoking internal agent logic procedures.
ALL encapsulates agent's internal application logic. It is partially automatically generated and it requires developer's further intervention to implement agent's internal procedures that will handle incoming messages and create outgoing messages during conversation progress, based on the agent's action definition in the scenario description(s).

5. Conclusion

Complex interactions within all but most simple multi-agent systems present a serious challenge during system design, implementation, testing and runtime analysis. Several approaches addressing those challenges have been employed so far and their strengths and weaknesses are outlined in this text. A scenario-based approach to modeling interactions in multi-agent systems is described, based on a notion of sequences of conversation actions, grouped into scenarios, describing conversational behavior of interacting agents. Scenarios are described by system designers using a proposed SDLMAS declarative language. The effect of using SDLMAS language and platform should be reflected in the significantly reduced effort invested during design and development of the communication aspect of a new multi-agent system, as well as in maintenance phase. A new version of SDLMAS allows for both linear and non-linear conversation sequences (loops), as well as increased runtime behavior control using metadata specification within scenario definitions.
Valid scenario models, resulting from processing of SDLMAS descriptions, are used for executable agent code generation for supported target agent platforms. SDLMAS Runtime framework provides runtime support for execution of SDLMAS based multi-agent systems and for gathering runtime behavioral data and its off-line analysis for profiling and optimization purposes.

6. References

Bratman, M. E., Israel, D. J., Pollack, M. E. (1988). Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4 (1988), pp. 349-355.

Cavrak I., Stranjak, A., Zagar, M. (2009). SDLMAS: A Scenario Modeling Framework for Multi-Agent Systems. *Journal of Universal Computer Science*, Vol. 15, No.4, (June 2009), pp. 898-925.

Cost, R., Chen, T., Finin, T., Labrou, Y., Peng, Y. (1999). Modeling Agent Conversations With Colored Petri Nets. *Proc. Workshop on Specifying and Implementing Conversation Policies*, Seattle, USA (1999), pp. 59-66.

Dinkloh, M. Nimis, J. (2003). A Tool for Integrated Design and Implementation of Conversations in Multiagent Systems. *Proc. AAMAS03 PROMAS Workshop on Programming Multi-Agent Systems Selected Revised and Invited papers*, Melbourne, Australia (2003), pp. 187-200.

Doi, T., Tahara, Y., Honiden, S. (2005). IOM/T: An Interaction Description Language for Multi-Agent Systems. *Proc. 4th International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS'05), Utrecht, Netherlands (2005), pp. 778-785.

Donikian S., (2001). HPTS: A Behaviour Modelling Language for Autonomous Agents, *AGENTS'01*, May 28 - June 1, 2001, Montreal, Quebec, Canada

Endriss, U., Maudet, N., Sadri, F., Toni, F. (2003). Protocol Conformance for Logic-based Agents. *Proc. 18th International Joint Conference on Artificial Intelligence* (IJCAI-2003), Acapulco, Mexico (2003), pp. 679-684.

Estava, M., Rodriguez, J. A., Sierra, C., Garcia, P., Arcos, J. L. (2001). On the Formal Specifications of Electronic Institutions. In *Agent Mediated Electronic Commerce*, The European AgentLink Perspective, Lecture Notes In Computer Science vol. 1991. Springer-Verlag, London (2001), pp. 126-147.

FIPA: Foundation for Intelligent Physical Agents. Available at: http://www.fipa.org

Gomez-Sanz, J. J., Fuentes, R. (2003). Agent Oriented Software Engineering with INGENIAS. *Proc. 3rd International Central and Eastern European Conference on Multi-Agent Systems CEEMAS 2003*, Prague, Czech Republic (2003), pp. 394-403.

Greaves, M., Holmback, H., Bradshaw, J. (2000). What Is a Conversation Policy? In *Issues in Agent Communication*, F. Dignum and M. Greaves, Eds. Lecture Notes In Computer Science, vol. 1916. Springer-Verlag, London, UK (2000), pp. 118-131.

Huget, M. P. (2002). A Language for Exchanging Agent UML Protocol Diagrams". Technical Report ULCS-02-009, The University of Liverpool, Computer Science Department, UK (2002).

Ishida, T., Q. (2002). A Scenario Description Language for Interactive Agents. *Computer*, 35, 11 (2002), pp. 42-47.

JADE: Java Agent Development Framework. Available at: http://jade.cselt.it

Jennings, N. R., Wooldridge, M. (1999). Agent-Oriented Software Engineering. *Proc. 9th European Workshop on Modeling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering* (MAAMAW-99), Valencia, Spain (1999), pp. 1-7.

Milner, R., (1989). Communication and Concurrency. Prentice-Hall International (1989).

Paurobally, S., Cunningham, J. (2003). Achieving Common Interaction Protocols in Open Agent Environments. *Proc. 2nd international workshop on Challenges in Open Agent Environments* (AAMAS 03), Melbourne, Australia (2003).

Purvis, M. K., Cranefield, S., Nowostawski, M.,Ward, R., Carter, D., Oliviera, M. A. (2002). Agent Cities Interaction Using the Opal Platform. *Proc. Workshop on Challenges in Open Agent Systems, The 1st International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS02), Bologna, Italy (2002).

Quenum, J. G., Aknine, S., Briot, J-P, Honiden, S. (2006). A Modelling Framework for Generic Agent Interaction Protocols. *Proc. 4th International Workshop on Declarative Agent Languages and Technologies*, Hakodate, Japan (2006), pp. 207-224.

Richters, M., Gogolla, M. (1998). On Formalizing the UML Object Constraint Language. *Proc. 17th International Conference on Conceptual Modeling*, Singapore (1998), pp. 449-464.

Stranjak, A., Dutta, P. S., Ebden, M., Rogers, A., Vytelingum, P. (2008) A Multi-Agent Simulation System for Prediction and Scheduling of Aero Engine Overhaul. *Proc. 7th International Conference on Autonomous Agents and Multiagent Systems: industrial track* (AAMAS2008), Estoril, Portugal (2008), pp. 81-88.

Shinoda, K., Noda, I., Ohta, M. (2003). Application of Parallel Scenario Description for RoboCupRescue Civilian Agent. *RoboCup 2003 International Symposium*, Padua, Italy July 10-11, 2003.

Shirai, T., Takano, M., Miyahara, H., Tajima, K., Kandori, K., Shimojo, S. (1999). Agent Enabled Scenario Language for Production of Interactive TV Program. *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing* (PACRIM 1999), 22 Aug 1999 - 24 Aug 1999.

Walton, C. D. (2003). Multi-Agent Dialogue Protocols *Proc. 8th International Symposium on Arti_cial Intelligence and Mathematics*, Fort Lauderdale, Florida (2003).

Warmer, J., Kleppe, A. (1999). OCL: The Constraint Language of the UML Journal of Object-Oriented Programming (1999), pp. 10-13.

Winikoff, M. (2005). Towards Making Agent UML Practical: A Textual Notation and a Tool *Proc. 5th International Conference on Quality Software* (QSIC'05), Melbourne, Australia (2005), pp. 401 - 412.

Wooldridge, M. (2000). Semantic Issues in the Verification of Agent Communication Languages. *Autonomous Agents and Multi-Agent Systems*, 3, 1 (2000), pp. 9-31.

Wood, M. F., DeLoach, S. A. (2000). An Overview of the Multiagent Systems Engineering Methodology *Proc. 1st International Workshop on Agent-Oriented Software Engineering*, Limerick, Ireland, 2000., pp. 207-221.

Wooldridge, M., Jennings, N. R., Kinny, D. (2000). The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems Archive*, 3, 3, pp. 285-312.