ABSTRACTION FROM COLLABORATION BETWEEN AGENTS USING ASYNCHRONOUS MESSAGE-PASSING

Bent Bruun Kristensen
Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Campusvej 55, DK-5230, Odense M, Denmark

Keywords: Abstraction, Associative Modeling and Programming, Asynchronous Message-passing, Collaborating Agents.

Abstract: Collaboration between agents using asynchronous message-passing is typically described in centric form distributed among the agents. An alternative associative form also by means of message-passing is shared between agents: This abstraction from collaboration is a descriptive unit and makes description of collaboration between agents simple and natural.

1 INTRODUCTION

Agents are autonomous, execute concurrently and communicate by means of synchronous or asynchronous message-passing (Scott, 2009). In a system executing agents at various times coordinate and typically communicate to exchange data. This collaboration between the agents can be described by different approaches. We focus on agents collaborating by means of asynchronous message-passing.

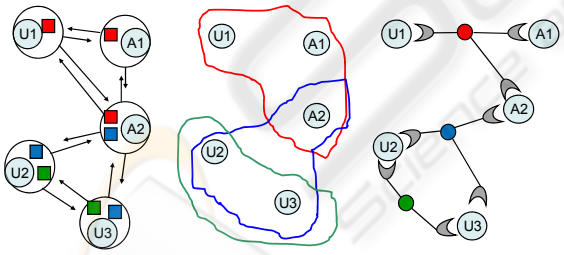

Figure 1: Collaboration: Understanding and Modeling.

The typical form of the message-passing description is *centric* in the sense that the message-passing constructs used to express the collaboration are placed in the individual code sequences of the agents. An alternative form of description is presented, namely to place message-passing mechanisms in an *associative* construct outside the agents and shared by individual agents. The two forms of description illustrated in Figure 1 (centric to the left and associative to the right) are characterized by a classic example, and evaluated.

Associations are abstractions from collaborations including communication, coordination and cooperation. The abstraction supports our understanding (Figure 1 in the middle) by modeling and programming collaboration as a unit: "Without abstraction we only know that everything is different" (Booch, 2007). Associations and collaborations are seen as concepts and phenomena and possess properties. Because an association is a descriptive unit collaboration may be described by simple clauses.

2 ASYNCHRONOUS MESSAGE-PASSING

We present concrete mechanisms for associative collaboration between message-passing agents. The semantics of the mechanisms is essential, but the syntax is only for illustrative purpose. Message-passing is illustrated by `Send(R, x)` — message `x` is sent to agent `R` (similar to "no-wait-send" (Scott, 2009)), and by `Receive(S)→y` — a message is received from agent `S` and assigned to `y` (similar to "polling without blocking" (Scott, 2009)).

2.1 Centric Form

Centric collaboration in schematic form is illustrated in Figure 2 where class `Sender` has a

Bruun Kristensen B. (2010).
ABSTRACTION FROM COLLABORATION BETWEEN AGENTS USING ASYNCHRONOUS MESSAGE-PASSING.
In *Proceedings of the 12th International Conference on Enterprise Information Systems - Software Agents and Internet Computing*, pages 86-92
DOI: 10.5220/0002898700860092
Copyright © SciTePress

reference R to an agent of class Receiver, a message x, and its action part. Similarly class Receiver has a reference S to an agent of class Sender, a message y, and its action part. We assume that agents SS of class Sender and RR of class Receiver exist such that SS's R reference refers to RR and RR's S reference refers to SS: By Send(R, x) agent SS sends the message x to agent RR. And by Receive(S)→y agent RR receives a message from SS and assigns it in y. The communication between the agents SS and RR is asynchronous, i.e. in the communication illustrated the message send is not necessarily the message received.

```
class Sender                    class Receiver
      extends Agent {                 extends Agent {
   Receiver R                      Sender S
   Message x                       Message y
   …                               …
   Send(R, x)                      Receive(S)→y
   …                               …
}                               }
```

Figure 2: Centric: Asynchronous Message-Passing.

2.2 Associative Programming and Modeling

Object-oriented programming includes centric descriptions, and collaboration is implicitly described only and distributed among methods of autonomous objects. In object-oriented methodologies alternatives exist typically only for analysis and design, but not for implementation. Associative programming and modeling (Kristensen, 2006) include:

- Associations support associative modeling and programming through abstractions from collaboration. An association is a descriptive unit of integrated collaboration and role aspects. Associations differ from usual classes because collaboration is between autonomous entities.
- The directive of an association (sequencing rules for interactions among the autonomous entities) is a central description related to the participating entities. The interactions are processed sequentially.
- An entity is autonomous: Only the entity itself may execute its methods. Action parts of entities (action sequence to be executed) execute concurrently.
- An entity executes its contributions (i.e. a method invoked by the entity) to the collaboration in the context of the entity. An entity participating in various associations executes contributions from the directives interleaved.

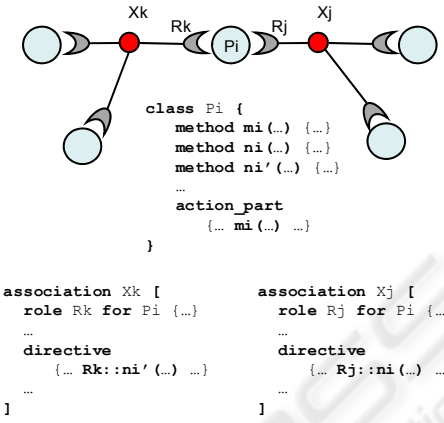

Figure 3: Associative Modeling and Programming.

Interleaved execution is illustrated in Figure 3: Associations Xj and Xk have roles for Pi named Rj and Rk and directives including Rj::ni(…) and Rk::ni'(…), respectively. Class Pi has methods ni, ni' and mi, as well as an action part including an invocation of mi. Assume (among others) that entity eP of Pi is engaged as roles Xj and Xk in instances of associations Xj and Xk. Assume that to eP is about to execute mi(…) and through the as roles Xj and Xk about to contribute with Rj::ni(…) and Rk::ni'(…), respectively. Then interleaved execution for eP in this schematic situation means, that exactly *one* out of mi(), ni(…) and ni'(…), is selected randomly and executed by eP. These actions except for the one selected remain ready to execute (possibly with additional actions form other associations) after the execution of the selected method for the following selection and execution by eP.

2.3 Associative Form

Here associations are between agents and enhanced by message-passing language constructs, but for simplicity reasons roles are not included as an integral part of associations. In message-passing associations the agents participating in associations are executing according to the above general description of associations. Associative collaboration is illustrated in Figure 4 where Association between Sender and Receiver describes a schematic collaboration between R and S. S sends the message available as x to agent R by S::Send(R, x), and R receives a message from S to be stored in y by R::(Receive(S)→y). The

communication between the agents is asynchronous, i.e. in the communication illustrated the message send is not necessarily the message received.

```
class Sender                    association Association [
      extends Agent {              Receiver R
   Message x                       Sender S
   …                               …
}                                  S::Send(R, x)
                                   …
class Receiver                     R::(Receive(S)→y)
      extends Agent {              …
   Message y                    ]
   …
}
```

Figure 4: Associative: Asynchronous Message-Passing.

Centric and associative collaboration between agents by means of asynchronous message-passing are illustrated in Figure 5. In the centric description (to the left) the interaction constructs are separated and specified in the action parts of the participating agents. Boxes represent agents taking part in execution and arrows represent agent references. In the associative description (to the right) communication is specified in the association on behalf of the agents. Boxes represent agents participating in associations and the oval with arrows represents an association, where the agents execute their contributions interleaved.

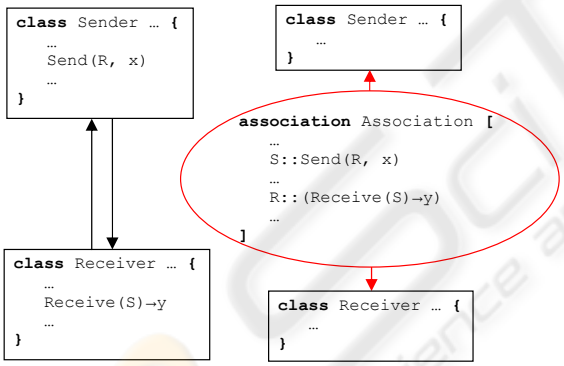

Figure 5: Collaboration: Centric and Associative.

2.4 Additional Coordination

The coordination of collaboration between `Sender` and `Receiver` does not ensure that the message send *is* the message received when `S` sends the message `x` to agent `R` by `Send(R, x)`, and `R` receives a message in `y` from `S` by `Receive(S)→y`: The message `x` may not be received or it is the first (not used so far) message received from `S`. To remedy this we include the method `AwaitMessage(…)` illustrated in Figure 6: All messages received are accumulated until a message from agent A has been received. The message may be received before or after `AwaitMessage(…)` is invoked, because a queue of received messages is maintained for each agent. The method `Receive()` is used to retrieve the next message received to illustrate the situation where the agent is responsible for retrieving its messages (`Receive()` is without parameters but else similar to `Receive(…)` with an agent as parameter). If no messages are available at a given time `waitAwhile()` makes the execution wait for a while (instead of e.g. introducing "blocking" and agent scheduling model).

```
… Message AwaitMessage(Agent a) {
   while (!getMessageSent(a)) {
      m = Receive();
      if (!m==null) putMessageSent(m)
      else waitAwhile();
   }
   return clearMessageSent(a);
}

… class MessageList {
   … Boolean getMessageSent(Agent a) {…}
   … void putMessageSent(Message m) {…}
   … Message clearMessageSent(Agent a) {…}
   …
}
```

Figure 6: `AwaitMessage(…)` and `MessageList`.

`MessageList` accumulates messages received by an agent and maintains a queue of received messages from each sending agent. The methods include `Boolean getMessageSent(Agent a)`: Check if a message with sender `a` is received, i.e. the queue for agent `a` is not empty; `putMessageSent(Message m)`: accumulate message `m`, i.e. add message to the queue; `Message clearMessageSent(Agent a)`: Remove message with sender `a` from accumulated messages, i.e. remove message from the queue. By `AwaitMessage(S)` we are sure that a message has been received from `S`, and that the first message received from `S` is returned.

3 BOUNDED BUFFER EXAMPLE

We describe the Bounded Buffer example by collaborating message-passing agents in centric and associative form.

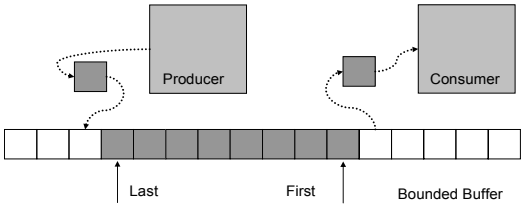

Figure 7: Illustration: Bounded Buffer Problem.

The Bounded Buffer problem is illustrated in Figure 7 where `Producer` produces artifacts and `Consumer` consumes artifacts—concurrently, but production times and consumption times are not related:

- `Producer` delivers each artifact produced to `Bounded_Buffer` and `Consumer` retrieves each artifact for consumption from `Bounded_Buffer`. `Producer` and `Bounded_Buffer` are coordinated during the transfer of an artifact—similarly for `Consumer` and `Bounded_Buffer`.
- `Bounded_Buffer` is bounded, i.e. a maximum number of elements may be kept in the buffer. If `Bounded_Buffer` is full no more elements may be added to the buffer and `Producer` has to wait for `Bounded_Buffer` not to be full. If `Bounded_Buffer` is empty no elements can be retrieved from the buffer and `Consumer` has to wait for `Bounded_Buffer` not to be empty.

3.1 Centric Version

```
class Producer
    extends Agent {
  Bounded_Buffer BB
  Message x
  method Produce(…) {…}
  …
  (loop
    Produce()→x
    AwaitMessage(BB)
    Send(BB, x)
  loop)
  …
}

class Consumer
    extends Agent {
  Bounded_Buffer BB
  Message y
  method Consume(…) {…}
  …
  (loop
    Send(BB)
    AwaitMessage(B)→y
    y→Consume()
  loop)
   …
}
```

```
class Bounded_Buffer … {
  Producer P
  Consumer C

  data Buffer …
  Message x, y
  method Empty() {…}
  method Full() {…}
  method Put(…) {…}
  method Get(…) {…}
  …
  (|(loop
      wait(Full())
      x→Put()
      Send(P)
      AwaitMessage(P)→x
    loop)
    ,
    (loop
      wait(Empty())
      Get()→y
      AwaitMessage(C)
      Send(C, y)
    loop)
    |)
    …
}
```

Figure 8: Bounded Buffer: Centric Version.

The centric solution is illustrated in Figure 8 where `Producer`, `Consumer` and `Bounded_Buffer` are agents each describing their individual action parts:

- `Producer` continuously produces and delivers an artifact to `Bounded_Buffer` and `Consumer` continuously receives an artifact from `Bounded_Buffer` and consumes it. `Bounded_Buffer` continuously either accepts or delivers an artifact given that the buffer is not full or empty, respectively.
- In `Bounded_Buffer` the construction `(| … , … |)` means concurrent execution of the two parts. Collaboration is described by the acknowledgement transfer (i.e. a message with no additional contents is communicated) in `Send(P)` in `Bounded_Buffer` and `AwaitMessage(BB)` in `Producer` succeeded by the message transfer of `x` in `Send(BB, x)` in `Producer` and `AwaitMessage(P)→x` in `Bounded_Buffer`.
- Similarly by the acknowledgement transfer in `Send(BB)` in `Consumer` and `AwaitMessage(C)` in `Bounded_Buffer` succeeded by the message transfer of `y` in `Send(C, y)` in `Bounded_Buffer` and `AwaitMessage(B)→y` in `Consumer`.
- Throughout the examples `wait(…)` means that the agent executing … waits until the result of this execution becomes false.

3.2 Associative Version

```
class Producer … {
  method Produce(…) {…}
  …
}

class Bounded_Buffer … {
  data Buffer …
  method Empty() {…}
  method Full() {…}
  method Put(…) {…}
  method Get(…) {…}
  …
}

class Consumer … {
  method Consume(…) {…}
  …
}
```

```
association ProducerBuffer
  Producer P
  Bounded_Buffer BB
  Message x
  (loop
    wait(BB::Full())
    x→BB::Put()
    P::Produce()→x
    P::Send(BB, x)
    BB::AwaitMessage(P)→x
  loop)
]

association ConsumerBuffer
  Consumer C
  Bounded_Buffer BB
  Message x
  (loop
    wait(BB::Empty())
    BB::Get()→x
    BB::Send(C, x)
    C::AwaitMessage(BB)→x
    x→C::Consume()
  loop)
]
```

Figure 9: Bounded Buffer: Associative Version.

The solution is illustrated in Figure 9 including agents `Producer`, `Consumer`, and `Bounded_Buffer`:

- `ProducerBuffer` and `ConsumerBuffer` are associations between `Producer` and `Bounded_Buffer` agents and `Consumer` and `Bounded_Buffer` agents, respectively. `Producer` and `Bounded_Buffer` have no action part but contribute to `ProducerBuffer` by executing `Produce` and `Full/Put`, respectively—similarly for `Consumer`, `Bounded_Buffer`, `ConsumerBuffer`, `Consume` and `Empty/Get`.
- `ProducerBuffer` describes the action cycle: `Bounded_Buffer` waits if full, `Bounded_Buffer` stores `x` as next message, `Producer` produces the contents of a message in `x`, and eventually transfers `x` from `Producer` to `Bounded_Buffer` by

`P::Send(BB, x)` succeeded by `BB::AwaitMessage(P)→x`.

- `ConsumerBuffer` describes the action cycle: `Bounded_Buffer` waits if empty, `Bounded_Buffer` retrieves next message to x, transfers `x` from `Bounded_Buffer` to `Consumer` in `BB::Send(C, x)` succeeded by `C::AwaitMessage(BB)→x`, and eventually `Consumer` consumes the contents of the message.

4 EVALUATION

Centric collaboration cf. Figure 10 (left) is characterized by

- Concurrency is described implicitly by individual agents `producer/consumer` and explicitly in `bounded_buffer`.
- Collaboration is described by several `Send(…)` and `AwaitMessage(…)` at different points and with different purposes in the action sequences of the individual agents. For example the collaboration between `Producer` and `Bounded_Buffer` is initiated by the acknowledgement transfer in `Send(P)` and `AwaitMessage(BB)` and only when this is established the actual message transfer takes place by `Send(BB, x)` and `AwaitMessage(P)→x`.

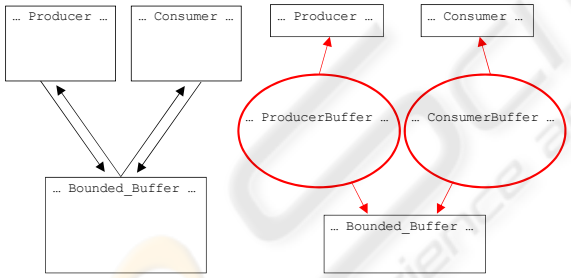

Figure 10: Bounded Buffer: Centric and Associative.

Associative collaboration cf. Figure 10 (right) is characterized by

- Concurrency is described by the different associations. Still the any contribution is executed by the respective agent.
- Collaboration is described by individual association units, in `ProducerBuffer` mainly by `P::Send(BB, x)` followed by `BB::AwaitMessage(P)→x` and in `ConsumerBuffer` mainly by `BB::Send(C, x)` followed by `C::AwaitMessage(BB)→x`. No additional sending and receiving to prepare for the actual sending and receiving a message is needed.
- Sequencing of contributions from the agents is described by concatenation of clauses in the directive of the association, i.e. in `ProducerBuffer` the clause `P::Send(BB, x)` is followed by `BB::AwaitMessage(P)→x` and in `ConsumerBuffer` the clause `BB::Send(C, x)` is followed by `C::AwaitMessage(BB)→x`.

The associative form is superior to the centric form with respect to modeling and programming collaboration because this form supports our natural understanding of collaborations between agents (in terms of `ProducerBuffer` and `ConsumerBuffer`) and because the abstraction captures collaboration as these descriptive units. The abstractions described are formed by our conceptualization of the system and are essential for understanding, modeling and communication (Booch, 2007). Alternatively, if the focus is on the behavior of the individual agents then the centric form may be preferred because the entire action part may be described as a unit.

In the descriptions the various elements of coordination appears differently in associative and centric forms. In the centric form concurrency appears natural by means of the action parts of the agents, whereas message-passing preparation and sending must be described explicitly by additional clauses. In the associative form coordination and message-passing appears natural whereas concurrency is naturally described by the association abstractions. Hence the associative approach is more simple, understandable and flexible than the centric approach.

4.1 Conceptualization versus Implementation

Associations based on asynchronous message-passing support our way of understanding collaboration between agents through abstraction, and the association functions as a natural language mechanism for describing systems. In addition by the association the agents collaborating have no references to each other (pointers/references considered harmful), i.e. they only know each other indirectly through the association. But because abstractions are descriptive units they appear as central descriptions. A system description is typically formed by a number of such abstractions, and these abstractions may be related through composition and specialization: The association is a

language aspect capturing our conceptualization of collaboration in a system.

Still associations are language mechanisms and not implementation specific technology. Associations may be implemented as central units similar but not identical to agents, but the actions of the directives are executed by the contributing agents. However the distribution of these contributions and sequencing of the directive itself may be maintained by such an implementation unit. The association controls its directive but the agents execute their contributions. Alternatively this control may be distributed among the agents. In order to support certain desirable conditions this implementation approach becomes decentralized. An association is then a shared plan with a current point of control. This plan is distributed to the next agent to contribute according to the plan. When an agent has completed its contribution, the agent is responsible for maintaining the plan and forwarding the plan further. The association is passive because the agents process the directive as a plan: The contributions from participating agents to the association are distributed to the agents for which to-do lists of actions are maintained and processed.

No matter if the association is implemented centralized or decentralized the idea of the association as a shared plan for collaboration makes it possible for agents precisely and understandably to explain their ongoing actions. The association works as a shared plan explaining not only what is going on but also why and in which context.

4.2 Experiment

The Bounded Buffer experiment is inspired from a project about transportation systems (FLIP) (Jensen, et al., 2005). The FLIP project investigated the process of moving boxes from a conveyor belt onto pallets and transporting these pallets. This process exists in the high bay area of the LEGO® factory with AGVs, no human intervention and only centralized control. A toy prototype inspired from this system (to bridge the gap between simulation and real physical applications) measured 1.5 by 2m, with three mobile robots (LEGOBots), two input stations, two output stations, one conveyor belt, and one station with empty pallets. The approach supported a fully distributed control for each LEGOBot. A LEGOBot was based on a LEGO® MindstormsTM RCX brick extended with a PDA and wireless LAN. The enormous problems with combining and maintaining the basic technology (including LEGO® MindstormsTM, RCX, PDA, WLAN) motivated the introduction of a virtual platform.

The Bounded Buffer experiment is based on a similar virtual platform illustrated in Figure 11. The experiments have several objectives including how to describe collaborations for classic problems like the Bounded Buffer example and how to implement the association abstraction, especially asynchronous message-passing. In Figure 11 the top part is a visualization of the Bounded Buffer example: Producer and Consumer are illustrated by respectively increasing and decreasing bars and Bounded Buffer is a queue of bars. The bottom part is the logical control illustrated in Figure 9. The logical control is an application framework in JAVA supporting `Association` and `Agent` as abstract classes. The simulator in the middle part consists of concurrently executing objects and is dynamically visualized. The objects of the logical model initiate and await the actual behavior in the simulator. The functionality of the simulator includes randomness etc. in order to expose relevant properties of a real physical system.

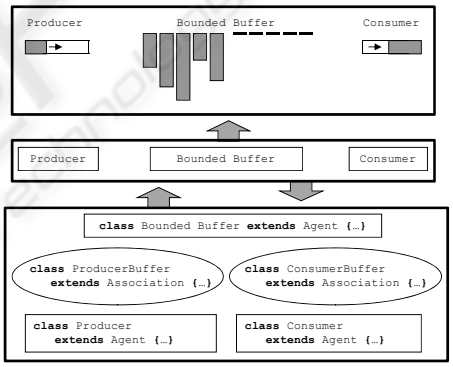

Figure 11: Experimental platform.

5 BACKGROUND

The specific characteristics are similar to synchronous and asynchronous message-passing in (Scott, 2009) whereas the basic agent and multi agent concepts are inspired from (Jennings & Wooldridge, 2000). The Java Agent Development Framework (JADE) (Bellifemine, et al., 2008) includes operations `send(…)` and `receive(…)`. Figure 12 illustrates these operations together with the operation `createReply(…)` that creates a new message `msgTx` as a reply to the message received, i.e. `msgRx`. In (Visual Studio, 2010) similar message-passing operations with varying synchronous and asynchronous aspects include

`send(…)`, `asend(…)`, `receive(…)` and `try_receive(…)`.

```
ACLMessage msgRx = receive();
if (msgRx != null) {
   System.out.println(msgRx);
   ACLMessage msgTx = msgRx.createReply();
   msgTx.setContent("Hello!");
   send(msgTx);
} else {
   block();
}
```

Figure 12: JADE Extract.

The association is a first class concept in modeling and programming notation (Kristensen, 2006). Various approaches to notation for non centric modeling and programming include: *Relations* (Rumbaugh, 1987) and associations in OMT (Rumbaugh, et al., 1991) are object-external abstractions but only for structural aspects. *Sequence* and *collaboration diagrams* in UML (Booch, et al., 1998) support the description of object interaction by means of method invocation. *Association = Activity + Role* (Kristensen, 2006) combines activities (Kristensen & May, 1996) and roles (Kristensen, 1995) in one abstraction supporting both roleification and execution. *Design pattern*s (Gamma, et al., 1994) capture experience of object oriented design and programming, but are only mental abstractions. Patterns for object collaboration include `DECORATOR`, `OBSERVER`, and `MEDIATOR`.

6 CONCLUSIONS

In the centric form of message-passing agents the focus is on the action sequence of the individual agent and the description of collaboration between agents is distributed among these. The associative abstraction is a descriptive unit and supports our natural understanding of collaboration as shared between agents. By means of the directive the description of collaboration becomes simple and natural.

Challenges for association based on asynchronous message-passing include

- Broadcast messages could be restricted to associations, i.e. only to agents participating in the ongoing collaboration.
- The facilities supported by an operation similar to `createReply(…)` could improve the expressional power of associations.
- In (JACK 2010) (agent oriented development environment and agent oriented extensions to JAVA) a message is received implicitly by the agent and an associated plan for handling the message may be initiated: An association could be seen as a similar plan for *several* collaborating agents.

ACKNOWLEDGEMENTS

We thank Palle Nowack at Alexandra Institute for inspiration and contribution.

REFERENCES

Bellifemine F. L., Caire G., Greenwood D., 2008. *Developing Multi-Agent Systems with JADE*. Wiley.

Booch G., Rumbaugh J., Jacobson I., 1998. *The Unified Modeling Language User Guide*. Addison Wesley.

Booch G., 2007. Private communication.

Gamma E., Helm R., Johnson R., Vlissides J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

JACK, 2010. Intelligent Agents — Agent Manual — Agent Practicals. AOS Group (Autonomous Decision-Making Software), *http://www.agent-software.com*.

Jennings N. R., Wooldridge M., 2000. Agent-Oriented Software Engineering. *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering*.

Jensen L. K., Kristensen B. B., Demazeau Y., 2005. FLIP: Prototyping Multi-Robot Systems. *Journal of Robotics and Autonomous Systems*. Vol. 53, pp. 230-243.

Kristensen B. B., 1995. Object-Oriented Modeling with Roles. *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*.

Kristensen B. B., May D. C-M., 1996. Activities: Abstractions for Collective Behavior. *Proceedings of the European Conference on Object-Oriented Programming*.

Kristensen B. B., 2006. Associative Programming and Modeling: Abstractions over Collaboration. *1st International Conference on Software and Data Technologies*.

Rumbaugh J., 1987. Relations as Semantic Constructs in an Object-Oriented Language. *Proceedings of the Object-Oriented Systems, Languages and Applications Conference*.

Rumbaugh J., Blaha J. M., Premerlani W., Eddy F., Lorensen W., 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

Scott M. L., 2009. *Programming Language Pragmatics*. Morgan Kaufmann Publishers.

Visual Studio, 2010. Visual C++ (Asynchronous Agents Library), *http://msdn.microsoft.com/en-us/library*.