TEMPORAL INTERACTION DIAGRAMS FOR MULTI-PROCESS ENVIRONMENTS

T. Y. Chen*, Iyad Rahwan**, Yun Yang*

* School of Information Technology
Swinburne University of Technology
PO Box 218, Hawthorn 3122, Australia
Tel: +61 3 9214 5505, Fax: +61 3 9819 0823
{tychen, yyang}@it.swin.edu.au

** Department of Information Systems
University of Melbourne
Parkville 3010, Australia
Tel: +61 3 8344 9236, Fax: +61 3 9349 4596
i.rahwan@pgrad.unimelb.edu.au

Abstract

This chapter introduces a novel notion of temporal interaction diagrams for distributed and parallel programming. An interaction diagram is a graphical view of computation processes and communication between different entities in distributed and parallel processes. It can be used for the specification, implementation and testing of interaction policies in distributed and parallel systems. Expressing interaction diagrams in a linear form, known as fragmentation, facilitates automation of design and testing of such systems. Existing interaction diagram formalisms lack the flexibility and capability of describing more general temporal order constraints. They only support rigid temporal order, and hence have limited semantic expressiveness. We propose an improved interaction diagram formalism in which more general temporal constraints can be expressed. This enables us of capturing multiple valid interaction sequences using a single interaction diagram.

Keywords: Interaction Diagrams, Interaction Scenarios, Fragmentation, Distributed Systems, Parallel Systems, Object Oriented Systems, Multi-Agent Systems.

INRODUCTION

Various attempts have been made to formalize interaction among computational entities, such as distributed, parallel, object-oriented systems and multi-agent systems (Hoare, 1985; Magee et al, 1995; Ronnquist & Low, 1996; Koskimies et al, 1996; Ronnquist & Low, 1997; Kinny, 1999; Chen et al, 2002; Bauer, 1999; Bauer et al, 2001). Traditionally, the system design stage involves a description of the steps taken in processing a particular task. In distributed systems, however, the implementation requires a clear picture of the separate computational threads of different processes. In parallel systems, it would be very helpful to be able to express the computation flow descriptions for different processes. Interaction diagrams are designed to support the representation and processing of these mingling activities. The linear representation of these diagrams facilitates the automation of the processes of diagram manipulation for design, report generation and testing. In particular, testing can be performed by comparing execution traces against specifications expressed in terms of interaction diagrams.

However, existing interaction diagram formalisms support quite rigid temporal order constraints only. An execution trace is said to be valid if it satisfies the interaction diagram. In other words, there is no way of specifying multiple valid traces without writing multiple versions of fixed execution traces. This can become a difficult job, especially in systems with sophisticated interactions. In such systems, the number of valid interactions can be quite large and there is a demand to more concisely express such flexibility in a single interaction diagram. Our research is a step towards achieving this goal.

In this chapter, we give an overview of existing interaction diagram formalisms and present an enhancement of a particular framework in such a way as to support more flexible interaction specification. The chapter is organized as follows. In the next section, we present an overview of an existing interaction diagram framework and introduce the basic notation to be used throughout this chapter. Then we introduce our novel extension. After that, an example demonstrating the merit of our framework is presented. Then we discusses current and future trends in interaction diagram frameworks. Finally, we conclude and summarise our ideas and results.

BACKGROUND

In this section, we present a particular interaction diagram model proposed in (Ronnquist & Low, 1996). The work described in this chapter is an extension of this framework. An interaction diagram is a graphical representation of the computation threads and communications in a distributed system and the like. It is a graph showing each process symbolically as one or more vertical bars, and the messaging between processes as horizontal arrows between these bars. A sample interaction diagram is shown in Figure 1, describing three processes A, B and C and the message sequences among them. There are a number of properties that are worth mentioning with respect to the meaning of this interaction diagram according to the formalism in (Ronnquist & Low, 1996).

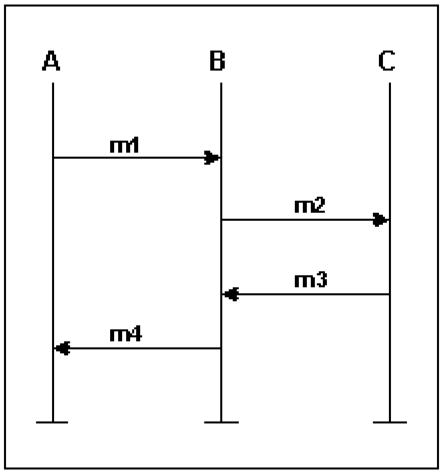

Figure 1. Simple Interaction Diagram

A *fragmentation* is an algebraic representation of an interaction diagram. In order to convert an interaction diagram into a fragmentation, we need to decompose it into its graphical elements, which correspond to *fragments*.. Figure 2 shows some types of these fragments. These fragments include the beginning of a process, the end of a process, a process sending a message, and a process receiving a message. Underneath every fragment is its corresponding algebraic form. Algebraic atoms can be used (forming a fragmentation) to describe a particular interaction diagram.

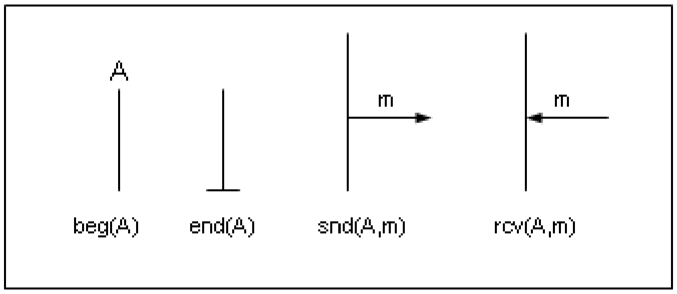

Figure 2. Graphical elements (*fragments*)

There are two types of fragmentation. However, we are only interested in *computation flow fragmentation* (Ronnquist & Low, 1996), which orders the fragments by grouping those of the same process together. Each process's fragments are ordered according to a top-to-bottom sequence (or temporal sequence). This represents the computation flow of a process. The computation flow fragmentations for Figure 1 are as follows.

For process A: <beg(A), snd(A,m1), rcv(A,m4), end(A)>

For process B: <beg(B), rcv(B,m1), snd(B,m2), rcv(B,m3), snd(B,m4), end(B)>

For process C: <beg(C), rcv(C,m2), snd(C,m3), end(C)>

According to Ronnquist and Low (1996), when actions concern a single process, the order of actions has significance for the interpretation of the diagram. The meaning of a single process computation flow is that the order in which the fragments occur reflects the order in which the corresponding actions take place. In other words, they state the **exact temporal order** in which the fragments or messages **should** take place. There is no mechanism to express the temporal constraints between different fragments. This is a severe limitation to the expressiveness of the framework. What we would like to achieve is a formal framework for supporting more flexibility in the temporal constraints.

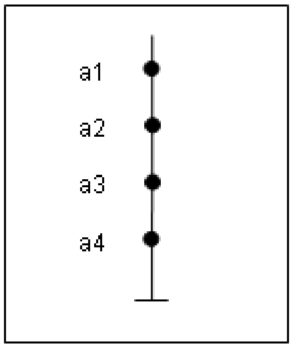

Figure 3. Single process diagram

In order to explain the limitation more clearly, we will give a simple example of one process. Suppose we would like to express the following temporal order constraints on the events of the diagram in Figure 3, where an event may be snd or rcv.

We will use a coffee making machine as an example for illustration. Suppose one process needs to perform the following tasks:

- First, send a message to the resources process requesting sugar and coffee (call this event a1).
- Then, receive a message from the resources process indicating sugar is ready (call this event a2). Also receive another message from the resources process indicating coffee is ready (call this event a3). It does not matter which of these acknowledgements occurs first, but they both need to be completed before moving to the next step.
- Now, send a message to the water process requesting hot water to be poured in the cup (call this event a4).

More abstractly, we have the following constraints:

- a1 must be before a2, a3 and a4
- the order of a2 and a3 is not important
- a4 must take place after both a2 and a3.

In fact, we can see the problem as follows. In a parallel system design and communication specification stage, instead of specifying a single valid execution sequence that must be followed, we may like to specify a number of valid sequences. Recall that interaction diagrams can be used for checking execution traces against specified order. In the example above, the designer's intention is to treat both event sequences a1, a2, a3, a4 and a1, a3, a2, a4 as valid. Using the old formalism, we need to provide two different (rigid) interaction diagrams. If the execution trace satisfies one of these diagrams, then the trace is correct. This technique becomes less practical in more complex settings where the number of possible valid sequences of event is large. In such a situation, a separate interaction diagram needs to be provided for each valid sequence, and checking needs to be carried out against all interaction diagrams until one (or none) matches. In the next section, we present an extension to the existing framework which supports more general and flexible constraints.

PROPOSED ENHANCEMENTS

In this section, we propose the enhancement of the current interaction diagram formalism as follows. We would like to distinguish between two types of temporal order constraints; namely, groups of events among which the order is important, and groups of events among which the order is not important.

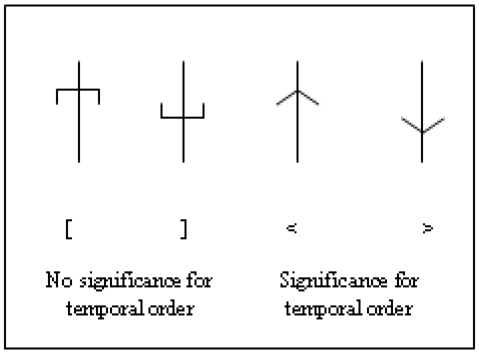
[
]
<
>
No significance for temporal order
Significance for temporal order

Figure 4. Ordering fragments

We will extend the set of graphical elements by including the fragments shown in Figure 4. A couple of fragments, corresponding to the '<' and '>' symbols (we will call them *angular brackets*), represent the start and end of a group of fragments among which the top-to-bottom order is the actual temporal order (that is as in the original formalism). This temporal order also states that, in addition to the specified temporal order, no external event is allowed to interleave with the enclosed events. The other couple of fragments, corresponding to the '[' and ']' symbols (we call them *square brackets*), represent the start and end of a group of fragments among which the top-to-bottom order does not necessarily reflect the actual temporal order. Throughout the paper, we will use the term "*bracket*" to refer to all types of brackets.

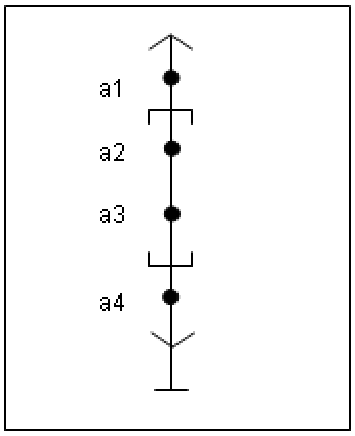

Figure 5. Extended single process diagram

A sample interaction diagram using some of the proposed fragments is shown in Figure 5. This interaction diagram expresses the temporal constraints mentioned in the previous section, which the previous formalism failed to express. Simply, the temporal order of the events within the square brackets are not important. However, event a1 must precede this group, and a4 must take place after it. The following is the corresponding modified computation flow fragmentation. (Note that a1, a2, etc, are placeholders for normal fragments such as snd and rcv).

< beg(A), a1, [a2, a3], a4, end(A) >

Note that this fragmentation is semantically equivalent to the following fragmentation (that is, changing the order of events inside the square brackets does not affect the meaning of the diagram):

< beg(A), a1, [a3, a2], a4, end(A) >

These fragmentations have the following properties:

- An opening fragment of a particular type should appear before any closing fragment of that type.
- At any stage of parsing the fragmentation, the number of opening fragments must be greater than or equal to the number of corresponding closing fragments of the same type.
- At the end of the interaction diagram (or the corresponding fragmentation), the number of opening and closing fragments should be equal (that is each opening fragment should have a corresponding closing fragment).
- Overlapping groups are not permitted, that is if a bracket of group g1 opens, and within that a bracket of group g2 opens, the closing bracket of group g2 must close before the closing bracket of group g1 does. That is each bracket is matched with the nearest corresponding bracket.
- An *entity* is either an atomic event or a group of entities enclosed within a matched pair of brackets. All entities are treated the same with respect to the higher-level group they belong to.
- The temporal relation between any two events is determined by the type of the nearest complete pair of brackets (angular or square) which contains them.

EXAMPLE AND OBSERVATIONS

In this section, we show another example to further illustrate our notation. We will exploit the supply chain automation domain. This domain is a typical example of situations in which there is a need for flexible specification of multi-process interactions. Suppose we have five processes:

- **Process A**: Representative of a personal computer (PC) manufacturing company. This company does not manufacture all computer parts, but rather purchases them from known partners (shown below). There are partners from which this party purchases motherboards, hard disks, and computer cases.
- **Processes B, C1, and D**: Representatives of manufacturers for PC motherboard, hard disks, and computer cases, respectively.
- **Process C2**: Representative of the manufacturing plant associated with process C1.

Figure 6 shows the interactions between the different processes, that is different messages passed between them. For simplicity, we do not include angular brackets at the beginning and the end of every process. The intuition behind the diagram with respect to process A is that before sending the order for computer cases, A first needs to order and receive motherboards and hard disks. This may be because it is very costly to store computer cases, and hence it would be more efficient to order them after all other components have been received. This way, computer assembly would take place as soon as the cases arrive. Intuitively, no deliveries should take place unless an order has been placed. This is reflected in the fact that no confirmation message can be received before the order. Moreover, ordering motherboards and ordering hard disks can take place in any order. That is, it does not matter which order process is performed first since assembly will not take place until the receipt of cases.

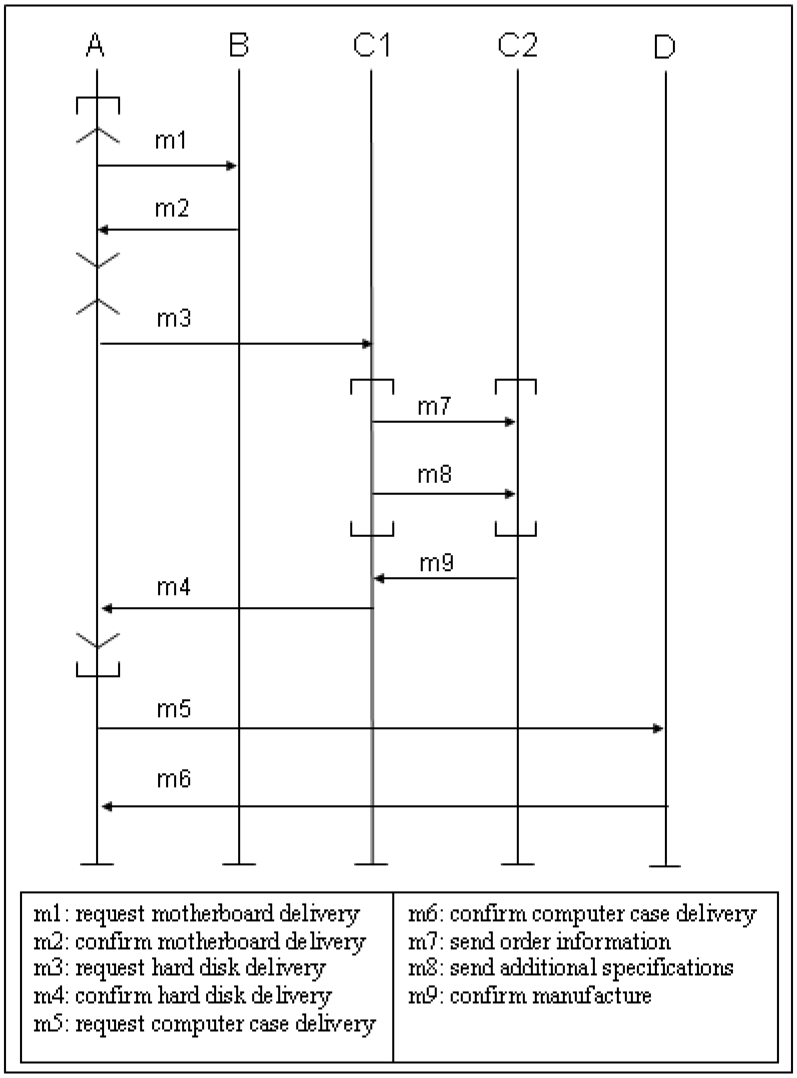
A
B
C1
C2
D
m1
m2
m3
m7
m8
m9
m4
m5
m6
m1: request motherboard delivery
m2: confirm motherboard delivery
m3: request hard disk delivery
m4: confirm hard disk delivery
m5: request computer case delivery
m6: confirm computer case delivery
m7: send order information
m8: send additional specifications
m9: confirm manufacture

Figure 6. PC manufacturing example

Now let us consider processes C1 and C2. After receiving a request for hard disk delivery through message m3, C1 must submit the order information to its manufacturing plant. C1 also needs to send additional technical specifications circulated internally from its design team (also represented by C1). It does not matter in what order these requests take place, but it is only after receiving both messages that the manufacturing plant can start manufacturing. This is why m9 can only be sent after receiving both m7 and m8. Note that according to the diagram, there is nothing to prevent messages m7 and m8 from being sent in one order and received in another. For example, C1 might send m7 followed by m8, but due to network problems, C2 might receive m8 before m7. Processes B and D only need to receive orders before they confirm delivery. The computation flow fragmentations of the system in Figure 6 are as follows.

<beg(A), [<snd(A,m1), rcv(A,m2)>, <snd(A,m3), rcv(A,m4)>], snd(A,m5), rcv(A,m6), end(A)>

<beg(B), rcv(B,m1), snd(B,m2), end(B)>

<beg(C1), rcv(C1,m3), [snd(C1, m7), snd(C1, m8)], rcv(C1, m9, snd(C1,m4), end(C1)>

<beg(D), rcv(D,m5), snd(D,m6), end(D)>

<beg(C2), rcv(C2,m7), rcv(C2, m8), snd(C2,m9), end(C2)>

Observation 1

Recall that the power of our framework stems from its ability to capture multiple valid execution or interaction sequences in a single diagram. Following are *some* interaction sequences that satisfy the diagram presented in Figure 6.

1. beg(A), beg(B), beg(C1), beg(C2), beg(D), snd(A, m1), rcv(B, m1), snd(B, m2), rcv(A, m2), snd(A, m3), rcv(C1, m3), **snd(C1, m7), rcv(C2, m7), snd(C1, m8), rcv(C2, m8)**, snd(C2, m9), rcv(C1, m9), snd(C1, m4), rcv(A, m4), snd(A, m5), rcv(D, m5), snd(D, m6), rcv(A, m6), end(A), end(B), end(C1), end(C2), end(D)
2. beg(A), beg(B), beg(C1), beg(C2), beg(D), snd(A, m1), rcv(B, m1), snd(B, m2), rcv(A, m2), snd(A, m3), rcv(C1, m3), **snd(C1, m7), snd(C1, m8), rcv(C2, m7), rcv(C2, m8)**, snd(C2, m9), rcv(C1, m9), snd(C1, m4), rcv(A, m4), snd(A, m5), rcv(D, m5), snd(D, m6), rcv(A, m6), end(A), end(B), end(C1), end(C2), end(D)

3. beg(A), beg(B), beg(C1), beg(C2), beg(D), snd(A, m1), rcv(B, m1), snd(B, m2), rcv(A, m2), snd(A, m3), rcv(C1, m3), **snd(C1, m8), snd(C1, m7), rcv(C2, m7), rcv(C2, m8)**, snd(C2, m9), rcv(C1, m9), snd(C1, m4), rcv(A, m4), snd(A, m5), rcv(D, m5), snd(D, m6), rcv(A, m6), end(A), end(B), end(C1), end(C2), end(D)

4. beg(A), beg(B), beg(C1), beg(C2), beg(D), **snd(B, m2), rcv(A, m2), snd(A, m1), rcv(B, m1),** snd(A, m3), rcv(C1, m3), snd(C1, m8), snd(C1, m7), rcv(C2, m7), rcv(C2, m8), snd(C2, m9), rcv(C1, m9), snd(C1, m4), rcv(A, m4), snd(A, m5), rcv(D, m5), snd(D, m6), rcv(A, m6), end(A), end(B), end(C1), end(C2), end(D)

Interaction sequences 2 and 3 differ from sequence 1 in the relative order for messages m7 and m8 which are highlighted in bold. Sequence 4 differs from sequence 3 in the order of the sending and receiving of messages m1 and m2 (that is in the order of the processes of ordering hard disks and motherboards). Note that the above are only examples of some valid sequences. Other valid sequences include all other combinations of orderings of interaction between process A and C1 with the interaction between C1 and C2. Also, processes can start and terminate in different orders as long as each process is created before it starts sending or receiving messages and terminates at the very end. For example, it does not matter which of beg(C2) or begin(D) takes place first.

We have just seen that using the, relatively simple, diagram presented in Figure 6, we are able to specify a large variety of valid interaction sequences among processes (or objects or agents). We do not have to specify multiple separate interaction diagrams to cater for all these sequences. This results in significant saving of time and space.

Observation 2

Note, that in the above fragmentation, the process of ordering motherboards must completely finish (that is the request as well the reply must both take place), before the process of ordering hard disks starts. This seems unnatural. One alternative approach is to replace the following notation:

[<snd(A,m1), rcv(A,m2)>, <snd(A,m3), rcv(A,m4)>]

with the following:

< [snd(A,m1), snd(A,m3)], [rcv(A,m2), rcv(A,m4)] >

which means that sending out the first two orders can happen in any order, and receiving the corresponding responses may also happen in any order, with the only condition being that *both* orders should be sent out before any order gets received. In reality, on the other hand, we might want to express that they can both happen in any order as long as no reply occurs before a request without disallowing a response to be received before the second order is sent. This is one of the limitations of this formalism. This is due to the fact that we do not allow interleaving between events belonging to different bracket pairs. We treat all events within a bracket pair as a single entity with respect to all other events outside that pair.

CURRENT AND FUTURE TRENDS

Software engineering practitioners face many challenges in the twenty first century. With the rate at which software systems are scaling up, and with the necessity of extensive interaction among distributed software and computer systems, there is an urgent need for sound software engineering frameworks that allow people to deal with such complexity. In distributed systems, it is very important to have efficient tools for the design, manipulation, deployment and testing of distributed computational entities. In Koskimies et al (1996), for example, ‘scenario diagrams’ have been used to study the interactions among object-oriented systems in dynamic object models. SCED (Koskimies et al, 1998) is a tool for using scenario diagrams for specifying object systems and generating state machines for describing the behaviour of each object from the scenario diagram specification.

Low et al (1997) presented the interaction diagram framework on which this work was based. They have built a tool for specifying interaction diagrams among multiple computational threads. They showed how an interaction diagram may be represented in different types of symbolic notations (called fragmentations). They presented a tool for translating an interaction diagrams into fragmentations and vice versa.

Interaction diagrams also have many important applications in multi-agent systems (Jennings, 2001; Wooldridge et al, 2000; Wooldridge, 1999). Jennings and Wooldridge (1998) proposed that an *intelligent* agent is capable of social behaviour; that is, capable of communicating with other agents in the system. In this context, there is a need for formalising such interaction in multi-agent systems. Such formalisation would be useful for designing and testing multi-agent systems, as well as visualising the computation flow of each agent and communication among agents.

Bauer et al (2001) presented an extension of the unified modelling language (UML), a de facto standard for object-oriented analysis and design, by adapting it for modelling protocols in multi-agent interaction, resulting in the Agent UML framework. Huget (2002a) used this framework to specify interaction in a supply chain management domain. Furthermore, Huget

(2002b) presented a framework for communicating UML interaction protocol diagrams among agents using an XML-based language.

We envisage many important development in the field of interaction frameworks in the next few years. Frameworks will emerge to facilitate different levels of interaction among computer systems. There is an urgent need for standardising methodologies for specifying interaction such that interaction specifications become more portable. Algorithms need to be put into place for manipulating interaction diagrams and for checking the validity of interaction traces. Furthermore, there will be a need for formal verification of the computational properties of those algorithms.

CONCLUSION

Existing interaction diagram formalisms and their corresponding linear notations (known as fragmentations) require different strict interaction diagrams for different execution sequences. This is because each interaction diagram is capable of expressing only a single, strict valid execution sequence. In this chapter, we have presented a formalism for describing flexible interaction diagrams which are able to express many possible execution sequences using one interaction diagram. Our formalism describes interaction diagrams which have a mix of two types of temporal relationships between events in a process, namely ordered and unordered temporal relationships. An ordered temporal relationship means that events should take place in the order specified while an unordered temporal relationship means that the order of executing two events is irrelevant. We have shown how combinations of these two types of relationships could be used to represent more complicated scenarios through a linear fragmentation.

REFRENCES

Bauer, B. (1999). Extending UML for the specification of agent interaction protocols. OMG document ad/99-12-03, FIPA submission to the OMG's Analysis and Design Task Force (ADTF) in response to the Request of Information (RFI) entitled "UML2.0 RFI".

Bauer, B., Müller, J. P., & Odell, J. (2001). Agent UML: A formalism for specifying multi-agent software systems. *International Journal of Software Engineering and Knowledge Engineering, 11*(3), 207-230.

Chen, T. Y., Rahwan, I., & Yang Y. (2002). Temporal interaction diagrams. In *Proceedings of the 2002 Information Resource Management Association International Conference* (pp. 843-846), Seattle, USA.

Hoare, A. (1985). *Communicating sequential processes*. Prentice Hall.

Huget, M. P. (2002a). An application of agent UML to supply chain management. To appear in *Proceedings of Agent Oriented Information System*, Bologna, Italy.

Huget, M. P. (2002b). Extending agent UML protocol diagrams. To appear in *Proceedings of Agent Oriented Software Engineering*, Bologna, Italy.

Jennings, N. R. (2001). An agent-based approach for building complex software systems. *Communications of the ACM, 44*(4), 35-41.

Jennings, N. R., & Wooldridge, M. (1998). Applications of intelligent agents. In N.R. Jennings & M. Wooldridge (Eds.), *Agent Technology: Foundations, Applications, and Markets* (pp. 3-28). Springer-Verlag.

Kinny D. (1999). The AGENTIS agent interaction model. In *Proceedings of the 5th International Workshop on Agent Theories, Architectures, and Languages* (pp. 331-344), Paris, France.

Koskimies, K., Männistö, T., Systä, T., & Tuomi, J. (1996). On the role of scenarios in object-oriented software design. In *Proceedings of Nordic Workshop on Programming Environment Research* (pp. 53-7), Aalborg University, Denmark.

Koskimies, K., Männistö, T., Systä, T., & Tuomi, J. (1998). Automated support for modeling of OO software. *IEEE Software, 15*(1), 87-94.

Low, C. K., Ronnquist, R., & Chen, T. Y. (1997). An automated tool (IDAF) to manipulate interaction diagrams and fragmentations for multi-agent systems. *International Journal of Software Engineering and Knowledge Engineering, 9*(1):127-149.

Magee, J., Dulay, N., Eisenbach S., & Kramer J. (1995). Specifying distributed software architectures. In *Proceedings of EsEC'95, Lecture Notes in Computer Science, 989*, 137-153.

Ronnquist, R., & Low, C. K. (1996). Formalisation of interaction diagrams. In *Proceedings of the 3rd Asia-Pacific Software Engineering Conference* (pp. 318-327), Seoul, Korea.

Ronnquist, R. & Low, C. K. (1997). Using interaction diagrams for multi-agent systems. In *Proceedings of the Joint 1997 Pacific Asian Conference on Expert Systems/Singapore International Conference on Intelligent Systems* (pp. 52-59), Singapore.

Wooldridge, M. (1999). Intelligent agents. In G. Weiss (Ed.), *Multiagent Systems* (pp. 27-78). MIT Press.

Wooldridge, M., Jennings, N. R., & Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems, 3*, 285-312.