Multi-Agent System Specification using TCOZ

Tim Miller and Peter McBurney*

Department of Computer Science,
The University of Liverpool, Liverpool, L69 7ZF, UK
{tim,peter}@csc.liv.ac.uk

Abstract. TCOZ is a specification language that combines the strengths of Object-Z and Timed CSP with the goal of specifying distributed systems containing objects that act independently and concurrently. Such goals are similar to the goals of the autonomous agent paradigm, in which agents are entities in an environment that act independently of one another, concurrently, and work proactively to achieve certain goals. This paper discusses the suitability of several TCOZ constructs in specifying multi-agent systems, and presents a small, yet non-trivial case study of a multi-agent system specification using TCOZ.

1 Introduction

Autonomous agents are software entities that have control, to a greater or lesser extent, over their own execution. With the recent rise of the Internet and of distributed computing systems, the autonomous agent paradigm has become important in designing, understanding and managing complex computer systems.

While much research into formal agent logics and languages is focused on modelling beliefs, desires, intentions, and knowledge of both individual agents and groups of agents, [12] languages for specifying and designing agent systems have received far less attention. In addition, much of the research in this area has focused on logic programming languages for agents, such as AgentSpeak(L) [8], or extending existing graphical languages to include the agent paradigm, such as the Agent UML [1]. Our focus is on the formal specification and verification of entire multi-agent systems and the environment in which they operate, with the aim that these specifications can be used as a starting point for system design. A formal specification of a system provides a precise and unambiguous description of the system's behaviour, and can serve many purposes, for example: providing a starting point for the design and implementation of the system; allowing developers to prove that certain properties hold or certain properties are achieved; and providing a starting point for test case and test sequence generation.

* The authors are supported by the *Personalized Information Platform for Health and Life Sciences (PIPS)* Project (EC-FP6-IST-507019) and from the UK EPSRC *Market Based Control of Complex Computational Systems* Project (GR/T10657/01), respectively.

There are examples of agent specification languages that provide good support for specifying agent systems, including the interactions between agents. AgentZ [2] is an agent-oriented extension to Object-Z, which includes concepts such as roles, agents, and environments. The OZS notation [5] is a hybrid of Object-Z and statecharts. Object-Z is used to specify the states and operations of an agent, and statecharts are used to specify the reactive properties of the agents. SLABS [13] is an agent-specific formal specification language that has constructs resembling classes and objects. While all of these languages are useful for modelling the states of agents, little support is provided for asynchronous, concurrent, autonomous behaviour. Concurrent METATEM [4] allows specification of agents using executable temporal logic, but state and composition are not as straightforward to model as the state-based approaches above.

In this paper, we discuss an existing language called *Timed, Communicating Object-Z* (TCOZ) [7], which is used for modelling real-time, concurrent systems by combining the strength of Object-Z [10], a state-based, object-oriented specification language, with Timed CSP [9], a real-time, concurrent language for modelling processes and their interactions. The goals of TCOZ (providing distributed, timed, concurrent, active objects) are similar to that of the agent paradigm, so we evaluate TCOZ as a specification language for multi-agent systems by specifying a small, yet non-trivial example of a multi-agent system for handling resource allocation.

2 TCOZ for Agents

In this section, we present some aspects of the TCOZ language, and discuss why we believe this is suitable for modelling software agents in a multi-agent environment.

2.1 Object-Z and State

Object-Z is an extension of the well-known Z specification language [11]. Among other things, Object-Z extends Z by the addition of a *class paragraph*, which resembles the class construct found in the object-oriented programming paradigm. A class consists of a state, which declares the state variables and their set of possible values, an initial predicate, which constrains the initial state value, and zero or more operations, which define transitions over the state.

Figure 1 shows the specification of a clock class, whose scope is defined by the box named *Clock*. The unnamed box in the class represents the state of class, and contains a variable $time : \mathbb{N}$, representing the current time. The *INIT* schema contains a predicate restricting the initial value of the clock's time to 0. The operation *Tick* increments the time by one, and the operation *SetTime* allows the environment to set the time via the input variable $time?$. Object-Z follows the style of Z by decorating input, output, and post-state variables with ?, !, and $'$ respectively.

Clock

$time : \mathbb{N}$	*INIT* $time = 0$
Tick $\Delta(time)$ $time' = time + 1$	*SetTime* $\Delta(time)$ $time? : \mathbb{N}$ $time' = time?$

Fig. 1. *Clock* Class in Object-Z

We can create an instance of *Clock* by using a variable declaration $c : Clock$. The commonly used "dot" notation is used to reference the state variables, for example, $c.time$, and to invoke operations, for example, $c.SetTime$. Operation variables can be renamed using the same notation as Z's variable renaming. For example, if we have a variable *newTime* in scope, we can set the time to the value of *newTime* like so: $c.SetTime[newTime/time?]$, which will substitute *newTime* for all occurrences of the name $time?$ in the operation.

These features of Object-Z are suitable for modelling the internal state of agents, such as the perceptions that an agent has of its environment. While it may be non-trivial to model beliefs about the environment, particularly nested beliefs (that is, beliefs about other agents' beliefs), many multi-agent systems, especially containing only software agents, need not model beliefs, and when they do, their beliefs are simple enough that they can be modelled using Object-Z.

2.2 Processes

Timed CSP [9] is an extension of CSP [6]. A specification of a process in CSP contains the set of events in which the process can take part, called the *alphabet* of the process, and the set of possible event traces that the process can perform, including communication between other processes (discussed further in Section 2.4). A specification of a system in CSP is the combination of those processes, which can be either synchronous or asynchronous. Throughout the section, we assume that P and Q are processes, a and b are events, and A is a set of events.

$a \to P$ is a process that is enabled when a occurs, and then behaves like process P. The environment can choose between two processes using the *external choice operator*: $\Box$. Using this, $a \to P \Box b \to Q$ specifies an event in which either a or b is chosen to occur by the environment, and then behaves as either P or Q respectively. The distributed choice operator allows the choice between an arbitrary number of events: $\Box\, a : A \bullet P(a)$ specifies the process in which an environment chooses an event in A to occur, and then behaves as the process $P(a)$.

Internal choice, which allows the process itself to make the choice, can be specified in the same way as external choice, except using the $\sqcap$ and $\bigsqcap$ operators respectively.

Two processes can be combined to behave in parallel asynchronously using the $|||$ operator, which specifies that $P \mid\mid\mid Q$ execute concurrently without any synchronisation. A distributed version of the operator is available, used like so: $\big|\big|\big|\, a : A \bullet P(a)$.

Processes provide an excellent way to model agents in a multi-agent environment, because they allow agents to have their own thread of control; something that is not possible using a state-based approach such as Object-Z.

2.3 Timing

Timed CSP extends CSP by adding the set $\mathbb{T}$, denoting the set of all moments of time, and timing primitives. The set $\mathbb{T}$ is a subset of the reals, so we can use the Z toolkit's arithmetic operators on time. For example, in the *Clock* class in Figure 1, we can replace all references to $\mathbb{N}$ with $\mathbb{T}$, while still using the arithmetic operator $+$.

TCOZ supports several of the Timed CSP timing primitives, but the only one relevant to this paper is $P \bullet \text{WAITUNTIL}\, t$, which enforces that if P takes less than t units of time to execute, then delay the process until t seconds has passed from the start of P. Other timing primitives include deadlines and events occurring at a particularly time.

This allows us to model timing properties in agents. While this is not so important in most multi-agent systems, it is not difficult to find examples in which timing is important. The case study in Section 3 is such an example.

2.4 Communication Channels

Communication between CSP process is achieved using *channels*. TCOZ allows channels to be declared in the state of an object. If a channel c is declared like so: $c : \mathbf{chan}$, then the event $c.v$ is defined as v being sent on channel c. Channels are untyped, and can therefore carry any expression. As with CSP, TCOZ uses the notation $c?v$ to specify that a process receives v on channel c, and $c!v$ to specify that a process sends v on channel c.

Using communication channels in TCOZ enables modelling of communication between agents. The object-oriented paradigm commonly discusses operation invocation as message passing, that is, invoking an operation on an object is defined as passing a message to that object. However, explicit communication channels provides agents with a way to pass messages to agents that are not visible to them, and a way of modelling the way the communication in the system will actually occur.

2.5 Active Objects

TCOZ allows the specification of two types of object: *passive* and *active*. Passive objects are objects that are controlled either by the environment or other objects in the system. Active objects have their own thread of control, and their operations are hidden from all other objects. Dong and Mahony [3] discuss active objects in detail.

Specifying active objects is achieved by declared a operation called MAIN. MAIN operations are non-terminating processes that define the behaviour of objects. For example, our clock example in Figure 1 can be improved such that it updates its own time by the addition of the following MAIN operation:

$$\mathrm{MAIN} == \mu\, C \bullet (\mathit{Tick} \bullet \mathrm{WAITUNTIL}\ 1s) \fatsemi C$$

In this definition, we use CSP's μ operator to specify a unique solution within the set of possible traces. The $\fatsemi$ operator in this definition is sequential composition of operations. So, the behaviour of the clock is to *Tick* once, and then wait until 1 unit of time has passed (*Tick* is constrained to take less than 1 unit of time), and recurse this behaviour. Any objects using this clock need not (cannot!) invoke *Tick* to update the time; any instances of *Clock* will keep time themselves.

Active objects are central to the reason that we have evaluated TCOZ for agents. One of the primary differences between objects and agents is that agents are autonomous. That is, they maintain their own thread of control, and do not necessarily act when asked to perform a task. Active behaviour is a way to operationalise autonomy. It also provides a way for an agent to be *proactive*. That is, it's behaviour is directed by the goals that it wants to achieve. By relying on other agents to send it messages, an agent can never be proactive.

3 Case Study

In this section, we present a case study of a small, yet non-trivial, multi-agent system specified in TCOZ. This case study is quite abstract, in that it models resources, tasks, and agents that reside in an environment, but with no detail of the resources and the types of tasks that are being performed. While the requirements of the system may not be the requirements that one would typically want in a system allocating resources (for example, the allocation is not guaranteed to be economically efficient), it provides us with a preliminary example of a system that contains the types of problems commonly found in multi-agent systems.

3.1 Requirements

There is a set of agents representing users (the users are not modelled in the system). These agents accept requests from their users, and only these users can submit requests to their agent. An agent can have more than one request submitted, but can deal with only one request at a time, and each agent has its own thread of control.

Each request contains a list of tasks that must be performed in order. Each task requires a resource for a non-zero amount of time. The execution of requests can be suspended, but tasks are atomic and their execution cannot be suspended. Thus, if the running time of a task is greater than an agent's allocation on the resource that requires that task, it must get access to that resource by asking the agent(s) that do have access. Each agent in the system is pre-allocated certain timeslots for each resource, with the agents taking turns for each resource. The pre-allocation is such that no agent has more than one resource at the one time, and that each agent is initially aware of the entire allocation.

During the running of the system, if one agent needs a resource, but it does not have access to that resource, then the agent that needs the resource may request it from the agent(s) that have control over it. These agents may choose to give up the resource, or may choose to keep it. We do not constrain how or why agents choose to give its allocation to a resource. No resource can be allocated to more than one agent at once, and it is assumed that it is in the interests of all agents to keep it this way. An agent will not give its allocation to another agent if it is using the requested resource at the specified time.

The allocation is not commonly known or stored, except in the initial state of the system. That is, when two agents agree to an exchange, they do not inform the other agents in the system. Therefore, the system is entirely distributed. If an agent believes that another agent has an allocation, it will request the allocation from that agent. However, it may already be allocated to another agent.

3.2 Resources

Resources are modelled using a given set called *Resource*. A given set is a set about which we know little other than it exists. The elements and structures are left unspecified.

3.3 Tasks and Requests

The *Task* class models the set of possible tasks. Each task requires a particular resource for a set amount of time, which is initially non-zero. The *Execute* operation models the execution of a task. At this level, we specify only that the task is completed, not what is executed. The variable δ is implicitly declared in every TCOZ operation. It denotes the amount of time that it requires for an operation to complete. An instance of the class *Task* is a passive object, and it is therefore the responsibility of the agents to tell the task the execute.

Task

$$resource : Resource \\ length : \mathbb{T}$$

INIT

$$length > 0$$

Execute

$$\Delta(length)$$

$$\delta = length \\ length' = 0$$

Requests are defined as being any non-empty sequence of tasks:

$$Request == \{r : \mathrm{seq}\ Task \mid r \neq \langle\rangle\}$$

3.4 Time

To keep time, we use a clock. This is similar to the *Clock* class presented in Figure 1, except that is includes the changes discussed in Sections 2.3 and 2.5.

3.5 Agents

As the control of this system is distributed among the system's agents, the agent definition is where most of the behaviour is modelled. Only one type of agent is presented, but the non-determinism specified means that the actual behaviour of agents can be quite different. The specification of the agent class is significantly larger and less trivial than the other classes in the system, as is often the case in a distributed system.

State. Firstly, we present the state of the agent. The state models five variables. *requests* the requests that the agent is trying to fulfill. *agents* is the set of agents in the system, with each agent having to be in its own set, as specified using the predicate $self \in agents$[1]. *alloc* is the agent's belief about the current allocation, which is most likely inaccurate, because the agent knows only of the changes made to the allocation in which it has been directly involved. The predicate part of the state declaration specifies that the agent's beliefs about an allocation must map allocations to agents that this agent knows about (that is, is in *agents*). c is a channel on which agents communicate with each other. Finally, *clock* is an instance of the *Clock* class. The agents' clocks are synchronised at the initial state of the system.

TaskAgent

$$
\begin{array}{l}
requests : \mathbb{P}\ Request \\
agents : \mathbb{P}\ TaskAgent \\
alloc : (Resource \times \mathbb{T}) \rightarrow TaskAgent \\
c : \mathbf{chan} \\
clock : Clock \\
\hline
self \in agents \\
\operatorname{ran} alloc \subseteq agents
\end{array}
$$

...

Initial Predicate. Firstly, each agent initially has no requests to fulfill. The second predicate specifies that no agent is allocated to two different resources at the same time.

INIT

$$
\begin{array}{l}
requests = \varnothing \\
\forall\, r_1, r_2 : Resource;\ t : \mathbb{T} \mid r_1 \neq r_2 \bullet alloc(r_1, t) \neq alloc(r_2, t)
\end{array}
$$

[1] *self* is an implicit object declared in each object's state that references itself.

Executing a Task. To execute a task, an agent must have enough time allocated on the required resource to execute it. We model the operation *CanExecute* as follows:

$$
\begin{array}{|l}
\hline
\textit{CanExecute} \\
\hline
\textit{task}? : \textit{Task} \\
\hline
\exists\, \textit{now} == \textit{clock.time};\ l == \textit{task}?.\textit{length};\ r == \textit{task}?.\textit{resource} \bullet \\
\quad \textit{now} \mathrel{..} \textit{now} + l \subseteq \{ t : \mathbb{T} \mid \textit{alloc}(r, t) = \textit{self} \} \\
\hline
\end{array}
$$

This operation is enabled if and only if the set of time units from now until the time when the task will finish executing is a subset of the time that this agent has allocated on the specified resource.

To model the execution of the task, we define an operation *ExecuteTask*:

$$\textit{ExecuteTask} == [\textit{task}? : \textit{Task}] \bullet \textit{CanExecute} \land \textit{task}?.\textit{Execute}$$

The $\bullet$ operator represents scope enrichment, which makes any variables declared on the left side of the $\bullet$ visible to the right side. The $\land$ operator specifies two operations executing concurrently if both are enabled, with any variables sharing the same name also share the same value. Therefore, *ExecuteTask* is enabled if *task*? can be executed, and the promoted operation *Execute* is executed on *task*?.

Sending and Receiving Offers. When an agent attempts to execute a task for which it does not have access to the appropriate resource for the required length of time t, it will ask the agents that do have that resource at any time of the next t time units to relinquish that resource. If and only if all the owners agree to this proposal, the agent executes the task using that resource. If they do not agree, the task is not executed. We model the behaviour of obtaining a resource in the operation *GetResource*:

$$
\begin{array}{l}
\textit{GetResource} == \textit{GetResourceOwners} \parallel [\textit{owners}? : \mathbb{F}\ \textit{TaskAgent}] \bullet \\
\quad (\bigwedge a : \textit{owners}? \bullet c!(\textit{self}, a, \textit{task}?) \to c?(a, \textit{self}, \textit{Agree}) \to \textsc{Skip}) \land \textit{ReAllocate}
\end{array}
$$

The operation *GetResourceOwners* outputs via a variable *agents*!, the set of agents that are allocated to the resource needed by a task. The $\parallel$ operator is the same as $\land$, but maps any occurrence of an output variable $v!$ to any occurrence of an input $v?$ on the other side of the operator, thus sharing the values of output values on one side with input values on the other. On the right side of the scope enrichment, we model the behaviour of the agent sending from itself to each owner, a message containing the task that it wants to execute. If all agents agree to relinquish the allocation for the time needed, by sending a reply *Accept*, then the agent reallocates the resources using the *ReAllocate* operation.

If any of the receiving agents do not choose to give up their allocation for the resource, then the agent can do nothing, because it cannot partially execute a task. This is modelled as follows:

$$FailResource == GetResourceOwners \parallel [owners? : \mathbb{F}\ TaskAgent] \bullet$$
$$(\bigsqcap a : owners? \bullet c!(self, a, task?) \to c?(a, self, Refuse) \to \textsc{Skip})$$

The left side of the scope enrichment is the same as in *GetResource*, but the right side models the case in which at least one agent refuses the proposal by sending the reply *Refuse*. Using these two operations, we redefine the *ExecuteTask* operation as follows:

$$\begin{aligned} ExecuteTask == [task? : Task] \bullet\ & CanExecute \land task?.Execute \sqcap \\ & GetResource \mathbin{;} ExecuteTask \sqcap \\ & FailResource \end{aligned}$$

This operation is enabled at all times. Either the operation can be executed because the agent has the appropriate allocation for the resource, OR, the agent gets access to the resource, and recursively calls *ExecuteTask*, which will only recurse once because the second time, the agent has access to execute to the task, OR, the agent fails to get the resource.

Clearly, we must model the behaviour of an agent receiving a reallocation request. This is modelled using an operation called *ReceiveRealloc*, and is similar to that of an agent sending a request, so we do not show the details here.

One advantage of using a language such as TCOZ to model this behaviour is that it adopts Object-Z's *blocking semantics*. That is, if the execution of an operation fails for any reason, then that operation was not enabled in the first place. This has many advantages, but none more so in that it allows us to abstract away from details about how to handle error states.

In this example, blocking semantics is particularly nice when the agent tries to get access to a resource. We model that the agent sends a message to every owner of that resource, and either they all agree, or at least one of them refuses. There is no need to model the process of the sending agent informing all agreeing parties that the deal is off for cases in which not all of the owners agree, which should be dealt with at the design level.

Executing Requests. Each agent can work on executing one request at a time. However, if an agent suspends the execution of a request, it can choose any of the other requests to continue with at any time.

An agent first has to choose one request from the set of requests:

$$NextRequest == \bigsqcap r : requests \bullet ExecuteRequest[r/r?]$$

It's important here that the choice is internal, because it is certainly non-deterministic, and which request is chosen must be up to the agent itself. Here, we make no restrictions on how the next request is chosen, so agents can choose a request that it can execute without reallocation. Once it is chosen, the agent uses the *ExecuteRequest* operation:

$$
\begin{aligned}
\mathit{ExecuteRequest} == {} & [r? : \mathit{Request} \mid \#r? = 1 \lor \mathit{head}(r?).\mathit{length} > 0] \bullet \\
& \qquad \textstyle\bigsqcap t : \mathit{Task} \mid t = \mathit{head}(r?) \bullet \mathit{ExecuteTask}[t/\mathit{task}?] \\
& \Box \\
& [r? : \mathit{Request} \mid \#r? > 1 \land \mathit{head}(r?).\mathit{length} = 0] \bullet \\
& \qquad \textstyle\bigsqcap r : \mathit{Request} \mid r = \mathit{tail}(r?) \bullet \mathit{ExecuteRequest}[r/r?]
\end{aligned}
$$

This operation is divided into two choices. The two options in this choice are disjoint, so it does not matter whether we use an external or internal choice operator. The first choice models the case in which the head task is yet to be executed (*length* > 0), or the head task is the only task in the request. In these cases, the *ExecuteTask* operation defined in Section 3.5 is used to try and execute the tasks. If the request is only one task long and the task has already been executed, *ExecuteTask* is still enabled, but has no effect. The second choice of *ExecuteRequest* is the case in which the request contains more than one task, but the head task has already been executed (*length* $= 0$). In this case, a recursive invocation is made on *ExecuteRequest* to execute the tail of the request.

Agent Behaviour. So far, we have shown parts of our specification for agents executing requests and getting access to resources. However, we want our agents to be autonomous, and as a result, these operations are not invoked by other objects in the system, and must be brought together into a MAIN operation.

To do this, we declare an operation *Go*, which models one "step" in the system. That is: receiving a new request, receiving an offer to give up a resource, or executing a part of a request. However, it is not as simple as conjoining all of these operations into one choice. We want to allow these behaviours to run concurrently, but we note that operations may not be enabled at all times. For example, an agent cannot receive a new request and execute a request simultaneously if there is no incoming request. Therefore, we must explicitly specify all of the combinations of operations, but the only time *NextRequest* is not used is when there are no requests to execute.

Once we have the *Go* operation, we put it together with the operation *Purge*, which removes any completed requests from the set of requests, and recurse over this behaviour.

$$\textsc{Main} == \mu\, \mathit{TA} \bullet \mathit{Go} \mathbin{;} \mathit{Purge} \mathbin{;} \mathit{TA}$$

3.6 The System

The system is declared as an active object that conjoins a set of *TaskAgent* instances, and runs them concurrently. The predicate part of the state declaration ensures that all agents share the set of known agents. The initial predicate ensures that at the start of the system, all agents share the same belief about allocations, all of the agents' clocks are synchronised, and each agent is in its initial state.

$$
\begin{array}{|l}
\hline
\textit{System} \\
\begin{array}{|l}
\hline
agents : \mathbb{P}\ TaskAgent \\
\hline
\forall\, a : agents \bullet a.agents = agents \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{INIT} \\
\forall\, a, b : agents \bullet a.alloc = b.alloc \land \\
\qquad a.clock.time = b.clock.time \\
\forall\, a : agents \bullet a.\mathit{INIT} \\
\hline
\end{array} \\
\textsc{Main} == \mathop{|||}\, a : agents \bullet a.\textsc{Main} \\
\hline
\end{array}
$$

3.7 Weaknesses

There are, of course, drawbacks our TCOZ model, mainly concerning undesirable behaviours. For example, it is possible that, because an agent initially has no requests to fulfill, it may continually execute *Idle*, never accepting any requests. Also, an agent may give up its allocation to resources too often, thus starving itself of access and not executing its requests. However, these are not great problems because we can assume that any implementation of these agents will be sensible. The most sensible implementation would interpret the $\sqcap$ operator in this particular definition of *Go* as "else-then". That is, try the first option. If this is not possible, then try the next, and so on, until *Idle* is only invoked if none of the others are possible.

Another weakness is with the de-centralised control of allocations. The system can come to a state in which most of the initial allocation has been changed, and because each agent only knows about the reallocations of which is was a part, many unnecessary messages are being sent throughout the system. One solution to this problem could be for all agents to broadcast, at set intervals, the allocations that they have. If all agents broadcast and read this information, then they will each have a complete and correct belief about the allocation.

4 Conclusion

In this paper, we discuss the use of TCOZ for modelling multi-agent systems. We believe that TCOZ, which combines the strengths of Object-Z and Timed CSP, provides many of the necessary constructs for specify multi-agent software systems, both at a level of individual agent specifications, and at a higher level of specifying interactions and cooperation between agents. These constructs include: state, which allows agents to record their aims and their perceptions of their environment; process primitives, which allow us to specify agents as having independent threads of control; communication channels, which allow us to specify the interactions between agents at a message passing level; and active objects, which allow us to specify agents as entities that are active and autonomous, and unable to be controlled by the environment in which they reside.

To evaluate TCOZ, we specified a non-trivial example of a multi-agent system being used to allocate resources. While TCOZ may not provide all of the necessary constructs to specify agents systems, it is clear that the goals of TCOZ (providing distributed, timed, concurrent, active objects) are similar to those of the agent paradigm.

4.1 Future Work

This preliminary paper leaves us with several important areas of future work:

- Investigate ways for TCOZ to support additional agent-specific constructs, such as roles and protocols.
- Investigate ways to model more complex agent states such as knowledge, beliefs, desires, and intentions, to increase the scope of possible agent systems that can be specified using TCOZ.
- Examine ways that libraries of complex interactions can be specified in TCOZ for use in specifications.
- Evaluate TCOZ's applicability to agent development frameworks.

References

1. B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction. In *Agent-Oriented Software Engineering*, pages 91–103. Springer: Berlin, Germany, 2001.
2. A.A.F. Brandao, P. Alencar, and C.J.P. de Lucena. Extending (Object-)Z for multi-agent systems specification. In *International Bi-Conference Workshop on Agent-Oriented Information Systems*, LNAI. Springer-Verlag, 2004.
3. J.S. Dong and B. Mahony. Active objects in TCOZ. In *Proc. of the 1998 IEEE International Conference on Formal Engineering Methods*, pages 16–25. IEEE Computer Society Press, 1998.
4. M. Fisher. A survey of concurrent MetateM — the language and its applications. In *Temporal Logic — Proc. of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag, 1994.
5. V. Hilaire, O. Simonin, A. Koukam, and J. Ferber. A formal approach to design and reuse agent and multiagent models. In *Agent Oriented Software Engineering*, volume 3382 of *LNCS*, pages 142–157. Springer, 2004.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
7. B. Mahony and J.S. Dong. Timed Communicating Object-Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, Feb 2000.
8. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer-Verlag, 1996.
9. S. Schneider and J. Davies. A brief history of Timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
10. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
11. J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall International (UK) Ltd, second edition, 1992.
12. M. J. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robotics and Autonomous Agents. MIT Press, Cambridge, MA, USA, 2000.
13. H. Zhu. SLABS: a formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, 11:529–558, Nov 2001.