Environment orientation: an architecture for simulating complex systems

Tim Hoverd[1] and Susan Stepney[1]

Department of Computer Science, University of York, UK, YO10 5DD
tim.hoverd@cs.york.ac.uk susan@cs.york.ac.uk

Abstract. A naïve implementation of a complex system simulation with its plethora of interacting agents would be to represent those interactions as direct communications between the agents themselves. Considerations of the real world that a complex system inhabits shows that agent interactions are actually mediated by the environment within which they are embedded and which embodies facilities used by the agents. This suggests an "environment oriented" simulation architecture.
Here we motivate and describe an abstract software architecture for an environment oriented approach to complex systems simulation, and sketch the implementation of this architecture in a number of different ways.

1 Introduction

Complex systems comprise of a number of agents that interact in some particular environment. The behaviour of any individual agent is relatively simple and local. A complex global behaviour emerges as a consequence the interaction of a large number of such agents in a particular environment.

A complex system can be simulated using computational devices to provide an executable model of the real world situation. Like all such models it should be constructed in manner that can feasibly be implemented, and may well avoid many real world details. However, such a model must encapsulate the key interactions between agents from which emerges the global behaviour.

2 Motivation

Complex systems get their emergent behaviour from interactions between the agents that comprise the system. Naïve implementation models therefore describe direct interactions between those agents.

Such an approach, however, leads to many implementation difficulties. Firstly, scaling the number of agents in a simulation to something representative of the modelled world is infeasible, because the number of communication channels required rapidly exceeds the capabilities of the simulation. Secondly, and of particular importance here, if such a model were to be implemented without detailed attention paid to concurrency issues, then it would doubtless deadlock

very quickly because of the loops apparent in the agent/channel graph. Consequently, such naïve implementations are never seen.

These deadlock issues are resolved in simulations by the introduction of techniques, such as the "client server" pattern for concurrent systems [12, 1] and barrier synchronisation [2], which impose a processing pattern onto communications between the various components of the simulation. These patterns seek to prevent the appearance of deadlocks.

In the case of the client server pattern components of the simulation are coded so as to operate in a manner reminiscent of "client-server" enterprise systems [21] or, more generally, multi-tier architectures [24]. In such systems the clients and the servers are layers in an architecture where the servers provide a pre-defined set of services to the clients. Each client is able to operate in a manner largely independent of others because the implementation of the system constrains the overall patterns of behaviour, for example by transactional access to an underlying repository [26], in such a manner as to guarantee various overall system properties.

Use of this convention introduces a pattern into the simulation that does not at first sight appear to exist in the real world being modelled. For example, the birds that flock above a city-centre park are not apparently working to some standard global pattern lest they deadlock and fall out of the sky. It appears that each bird is observing other birds and then doing what it wants, when it wants, and in whatever order it wants doing so in the sure and certain knowledge that its world really is deadlock free. That is, it appears as if the real world of these birds is rather different from a set of agents communicating with each other in a simulation of, for example, bird flocking behaviour.

The rest of this paper looks in more detail at what is really going on in a such a complex system, leading to some alternatives for the software architecture of complex systems implementations.

3 Real world agents and their environment

3.1 Action at a distance *versus* mediating fields

Let us think about how the real world agents actually interact. Although at first sight it is convenient to think about flocking birds interacting directly some thought shows that in fact this is a simplification of what is really going on.

A bird flying along reflects the ambient light into the space around it; as it sings it pressurises the air about it. Another bird, assuming that it is awake, is sensitive to the propagated light and air pressure, and in this way can both see and hear the first bird. That is, these two birds are not directly communicating with each other. The first is placing information into its environment, which information can be detected by the second bird when it observes its own environment, if it is interested in that sort of information.

Such a view, which is essentially an alternative model of interaction between the birds, relies upon a very detailed environment in which the agents, in this

case the birds, are embedded. One bird can always come along and look in the environment and see what another bird is frequently placing in the environment. A different bird might update the environment only seldomly, if it is just sitting quietly on some perch.

The real world is such a detailed environment; one where fields interact, photons pass each other and the rest of physics is implemented with ease. In this view the agents are *embodied* in the environment [19], and it provides services to those agents. Each agent just does what it wants without regard to direct interactions with other agents. That is, even in the real world, the agents in a complex system are interacting in a manner reminiscent of a client server architecture. The environment provides services to the agents, in a manner analogous to a *server*. The agents are *clients* of those services.

The naïve model of a complex system, with agents directly interacting with each other, is essentially "action at a distance". One agent must know directly what other agents exist that are interested in it and must directly interact with those agents. As this is happening those distant agents are also potentially interacting in the reverse direction.

Reflecting our observation of the real world, we instead take an "environment oriented" approach. Here agents do not interact directly, but communicate through some mediating fields that exist in their environment. In this approach there is no direct interaction at all; the lives of individual agents just affect each other by existing within the same set of fields; within the same physics.

As a different example of this "environment orientation" consider an adaptive immune system. Here the agents are the various molecules and cells that form the active components. The molecules are not directly signalling to each other about the feasibility of particular interactions. In this case an environment oriented model would represent these molecules by their concentrations in the environment which would affect the probability of interactions occurring as a consequence of the stochastic processes mediated by the environment.

3.2 State

The notion of "environment orientation" reflects the real world in a useful manner. However, what is it that agents communicate with the environment?

In something like a collection of birds flocking in the real world each bird has a large and complex internal state: it knows whether it is flying or not, how hungry it is, whether it needs to drink or defecate. But, from the point of view of flocking, other birds are interested only in the distances between the birds and what the perceived relative velocities of the other birds are.

That is, each agent has an "internal state" that represents everything it needs to know to behave appropriately. Further, each agent exposes an "external state" to the environment, which is available to other agents in the same environment. This external state could be simply a subset of the agent's internal state. For example, in the case of the bird it could just be that part of the internal state that represents the position and velocity of the bird. However, there are cases where the agent could deliberately mislead other agents with its external state.

For example, when one insect species mimics another it is deliberately creating external state to mislead observers about its internal state.

Complex system agents are essentially egocentric. That is, the emergent behaviour appears as a consequence of each agent just doing what it wants to do in its own environment. A flocking bird, then, does not know precisely where it is, just merely where other birds are relative to it. In a complex system simulation, something does need to know where the agents are, because those positions are the overall context of execution of the complex system. This context is the environment. That is, the environment must know where each agent, is and therefore the environment will know what other agents are in the vicinity of each agent. That is, the environment knows things about the agents that are not actually part of the agent's internal state. For example, a bird just thinks that it is flying in the direction of an interesting looking food source, but the environment knows that it is actually flying north-by-northwest.

A refinement to this notion is the observation that an agent generates some external state just by virtue of the physics of its environment. For example, photons just bounce off a bird, so other birds can see it, and are also able to infer position and velocity from those photons. This "involuntary" external state is contrasted with other state placed into the environment by an agent in a "voluntary" manner. Voluntary state could be, in the example of birds, a song that is sung in response to hearing the song of another bird of the same species (which it hears through the mediating environment), sung maybe for territorial enforcement or finding a mate.

3.3 Querying

In this environment oriented approach, each agent interacts with the environment to access information about the other agents' (external) states. Simplistically, each agent "asks" the environment for information about other relevant agents' state (the agents it can see, or hear, for example); this state information can then be used by the querying agent to update its internal state appropriately.

The reply to such a query is a set of values in some topology [7], which not only represents the set of all possible values but also describes how the values might change.

For example, in a bird flocking example, one of the items in a query result could represent a bird that is close to the querier. As such, the environment can accurately describe the (relative) position of the nearby bird and its velocity in terms of a three-dimensional Cartesian space. Furthermore, the topology of the particular space used might show that the nearby bird could move freely in the two horizontal dimensions but it was constrained to move only upwards in the vertical dimension because it is, at the moment, standing on the ground. That is, the reply to a query about the position of the bird gives a precise position in a space, but that space is further described by its extent and its shape.

If the bird being described is distant then the position of the bird may not be accurately described; for example, it might be clear in what direction the bird lies but its distance from the querier could be only poorly known. Similarly, the

velocity of the bird might be only poorly described, if at all, as the velocity of a distant agent which appears merely as a distant speck might be very hard to determine. In this case the reply is again a position in a space. However, in this case that space is two-dimensional being the surface of a portion of a sphere centred on the querying agent. Because the distance to the observed bird cannot be determined it cannot be moved inside or outside of that sphere.

Here the simulated environment is acting as the *embodiment* of sophisticated functions performed in the real world by both the agent itself and the environment. The agent itself detects the photons impinging on its retinas from a distant bird and attempts to calculate size, distance and velocity of the bird from those photons and, probably, experience in these sorts of situations. The real world, that is the environment, affects many aspects of the passage of those photons; it understands the albedo of the distant bird and can calculate how photons from the Sun are reflected by the bird, and how effectively those photons are transferred to the observing bird.

The environment oriented approach provides a way to separate concerns between the agents and their environment. In a particular simulation, the choice of what computation is performed by the environment, and what by agents, is a modelling decision. Certain functions may be embodied in the environment itself, and those calculations performed by the environment. Alternatively, responsibility for those those functions may be assigned to certain agents (either existing ones, or new ones designed to support those functions).

The notion of the results of the query being embedded in a topology allows the interaction between agents to follow a number of different patterns simultaneously. The example given above is a purely spatial one, the notion of space clearly being of significance in complex systems implementation as in [1]. However, the exact same query/response model could be used for any interaction between agents in a complex system. One extension of the simple spatial model is to note that a human agent is physically "near" to a collection of other human agents but may nonetheless communicate simply with other human agents whose telephone numbers are in the first agent's address book. That is, there are two sorts of "nearness" here: one is physical nearness, the other is "communicable" nearness. For some aspects of complex systems behaviour only the first sort of nearness would be relevant, for others both sets of "near" agents might be important. (This example is inspired by Milner's *bigraphical model* designed to model both a spatial and a connectivity configuration simultaneously [13].)

3.4 Environment orientation

In summary, the environment oriented approach to complex systems simulation eschews all representations of direct interactions between agents. Rather, all agent behaviour is seen as mediated through the environment within which all the agents are embedded; the essential rationale for this being that this is the way that the real world is structured.

Although the notion of the role of the environment is based on observations of the real physical world, the particular agents and behaviours that exist in a

simulation is a modelling decision. Each simulation should be constructed with the explicit knowledge of which aspects are to be embodied in the environment.

Regardless of its particular role, each agent has an internal state, representing what the agent knows of itself. It publicises some aspects of its state, its "external state" to the environment within which it is embedded. The agent may decide when to publicise its external state. Agent behaviour is provided for by allowing the agent to retrieve, from its environment, information about the external state of the agents with which it is interacting. Consequently, the environment must be aware of the agents with which each other agent can interact.

4 Software architectural styles

The "environment orientation" approach to complex systems must be readily implementable to be of use as an implementation platform for complex systems simulations. That is, we must define an abstract architecture that defines this sort of systems implementation.

The model as described is essentially a client server one. As has been described, real world complex systems are inherently "client server" in that the agents function essentially as clients of the environment.

A client server architecture is an appealing approach, since there is considerable experience with this approach that forms the basis of most high performance commercial computing. There are also several standardised abstract client server architectures, such as the REST architecture [4] that is the core of the Internet and the services it supports. These show the value of defining services in this manner.

The server in an "environment orientation" complex systems implementation must provide services that:

1. retain the external state of agents
2. provide that external state to other agents as and when required

The second of these services must reflect what aspects of each agents' external state is visible to a requesting agent. That is, the environment must know which other agents are in the "neighbourhood" of a requesting agent and must also know the topology of the result space in which to embed responses to requests.

In addition to providing such services to its clients, that is the agents, the environment may embody many aspects of the world that is being simulated or modelled. For example, if the complex system were modelling ant communication via stigmergy [3] then the environment itself could modify the external state of ant trails so that they decayed at the appropriate rate. This approach is a particular modelling decision. Alternatively, the ant trail might be modelled an agent; then it, and not the environment, would implement the process of pheromone decay.

Some aspects of this sort of architecture are seen in [1] where the implementation of various approaches to the representation of *space* in a complex system

```
while (true)
{
  Neighbourhood n = env.query(queryText,
                              <parameters drawn from internal state>)
  internalState.update(n)
  env.update(generateExternalState(internalState))
}
```

Fig. 1. Pseudo-code for agents using *query oriented* server

```
...
env.registerInterest(topic, callback)
...

void callback(Neighbourhood n)
{
  internalState.update(n)
  env.update(generateExternalState(internalState))
}
```

Fig. 2. Pseudo-code for agents using *subscription oriented* server

are investigated. The related "boids" simulation (based on [17]) uses a notion of "location" that is similar to the environment oriented server discussed here.

The first of these services listed above is susceptible to many different implementations, although the precise form of the delivered state is not defined here.

For the second service, there are two strategies, relating to a possible inversion of control. One approach would be for an agent that wishes to see the external state of a set of other agents, to make a *query* of the underlying "environment orientation" server. The query would provide the server will all the information it needed, along with its knowledge of the agents, to select the information required and provide it to the agents. This strategy, referred to here as *query oriented*, is summarised by the pseudo code in figure 1.

A complementary approach would be for agents to inform the server of the sort of information they were interested in, and to have that information delivered as and when it was available. In the meantime the agent would carry on with its normal behaviour. This strategy, referred to here as *subscription oriented*, is summarised by the pseudo code in figure 2.

These two approaches have different characteristics. The query oriented is appropriate for systems, perhaps like bird flocking simulation, where an individual agent can always be sure that its environment will change rapidly and apparently continuously. The subscription oriented approach would be useful for systems where some information was available only occasionally and unpredictably, or where it was needed to "interrupt" an agent from its normal activities. That

is, in situations where the particular environment was not changing apparently continuously.

In this abstract architecture, the server is the entire locus of inter-agent concurrency. That is, the agents execute without consideration for each other, simply relying on the server to provide pertinent information. This is the approach used in the world's largest commercial systems.

There are, though, at least two other issues that must be addressed here.

The first concerns that of fairness. If an environment server is being queried by a, potentially, very large number of clients then it must be the case that requests from those clients are handled in a fair manner. This is already an issue in multi-tier commercial systems and will not be further addressed here as it seems likely that existing approaches will satisfy the demands of a complex system simulation.

The second issue is that of time. Commercial systems are all "real time" systems, in the sense that the clients are usually aware of what the real world time is because that time is often pertinent to the processing that is being carried out. For a complex systems simulation there are further considerations. The simulation may run, as a whole, faster or slower than real time. In particular, individual agents can run at different rates from other agents, depending on how much processing they have to do (an active flying flocking bird will require more processing time to simulate unit time of its life than will an inactive perching sleeping bird). That is, the simulation as a whole, and the components of the simulation, are running in simulated time. As such, the "simulated time" is properly part of the environment within which the simulation's agents are embedded. Hence, an environment server should also provide a time service, that defines the current simulated time for each of the agents in the simulation. These agents can then, when necessary, consult the current time and use that to influence their activities.

5 Implementations

The architecture discussion so far has been devoid of implementation choices. The principal implementation choice is that of an environment server that can

- support the agents' external state where each item item of such state is in essence a tuple that contains whatever information is necessary for the particular application
- provide a means of accessing and distributing that state
- provide a mechanism for tracking the progress of simulated time

For example, in a bird flocking simulation each tuple retrieved by, or presented to, an agent would include another agent's relative position and perceived velocity. Additional entries in the tuple would allow the topology of the result space to be determined. For example, if the agent in question was distant then the perceived velocity might well be represented in a single-dimensional space

with very restricted possible changes instead of the three-dimensional space that would be appropriate for the velocities of nearby agents.

Regardless of these decisions, the data provided to a requesting agent takes the form of tuples. There are several possible implementation choices for how a server could provide the supply of tuples, described below.

5.1 Tuple spaces

The Linda programming language was first proposed in the mid 1980s [6] as a new way of handling concurrency and coordination. A running Linda system provides a "tuple space" which is populated, and examined, by a potentially large collection of concurrently executing agents. Linda provides primitives allowing the connected agents both to query the tuplespace for tuples that match some expression and to block waiting for an appropriate tuple to appear. As such the model supports both types of server architecture discussed in a straightforward manner.

The Linda concepts have been implemented in a number of modern programming languages. For example, JavaSpaces [11, 5] provides Linda-like facilities in the Java programming language as part of the Jini infrastructure. Rinda [18] provides tuplespaces for Ruby. TSpaces [8] is a simple implementation of the Linda ideas within Java from IBM.

A refinement of tuple spaces which is also relevant to this subject is that of tuple centres [16]. Tuple centres are essentially the notion of tuple spaces which have some behaviour. As such, a tuple centre could be seen as the implementation of a particular environment server.

5.2 Publish/Subscribe systems

The publish/subscribe pattern [25] is frequently supported by enterprise middleware, in particular by message oriented middleware [23]. For example, the Java Message Server [15] provides publish/subscribe facilities for users of the Java 2 Enterprise Edition. The publish/subscribe pattern provides for a server to distribute information on a number of *topics* to a number of connected clients. The pattern is often used, for example, in trading systems where some clients might require to be informed of changes in the prices of particular financial instruments when they occur. This is a very similar situation to that described as here as subscription oriented. A topic here could be, for example in the context of a bird flocking system, "the state of agents in the vicinity". Whenever one of those agents does indeed move the agent that registered the topic could be informed of a set of new tuples of information.

Publish/subscribe systems are used commercially in situations where there is a very high data rate, such as the instrument/price situation described above. As such they are also suitable for distributing information in a complex system simulation.

5.3 RDBMS

The use of a relational database management system (RDBMS) is a further possible implementation mechanism. Relational databases are essentially large containers for tuples. Each table in the RDBMS is a set of tuples with the same layout. Furthermore RDBMSs provide a highly expressive declarative query language (SQL [9, 10]) and are commonly used in situations where very high performance is required. As such they provide an attractive mechanism for the query oriented approach to the abstract architecture.

It is less clear how an RDBMS could be used for the subscription oriented architectural pattern. RDBMSs do support mechanisms that are capable of use in this manner (typically, triggers). However, they are clumsy in use and probably not suitable for the very flexible scenarios of complex systems.

5.4 Process oriented programming languages

Process oriented programming is at the heart of the CoSMoS[1] project (of which this work is part). The environment oriented architecture could be implemented using a process oriented language such as occam-π [20]. This is the language used for the models of space described in [1]. Using occam-π to implement simulations with the environment oriented architecture would ideally require the definition of a set of standard libraries that would hide many of the internal details, and allow the programmer to operate at a higher level of abstraction, purely in terms of things like tuples and queries.

6 Prototypes

We have implemented prototype complex systems simulations following the environment oriented architectural style. These prototypes have explored only the query oriented approach to the server. In particular, two prototype systems have been implemented, each of which is an implementation, in Java, of Reynolds' Boids [17], a very simple set of rules to simulate flocking.

As yet neither of the prototypes has been subjected to significant performance analysis and testing. In this first instance, we are simply establishing the capabilities of the abstract architecture.

6.1 Tuplespace prototype

The first prototype is an implementation using TSpaces (chosen due to the simplicity of configuring the server as compared with JavaSpaces).

The design of this system uses a single TSpaces server, running as a separate heavyweight process (a process running under control of the operating system and isolated from other such processes). A single boids heavyweight process implements each boid with a separate thread (a lightweight process not isolated

[1] http://www.cosmos-research.org

```
while (true)
{
  Neighbourhood n = env.query("allBoids")
  Vector acceleration = n.centreOfMassRule() +
                        n.matchVelocityRule() +
                        n.repelBoidsRule()
  velocity = velocity + acceleration
  env.updateTuple(boidId, velocity)
  wait(short_delay)
}
```

Fig. 3. Pseudo-code for Reynolds' boids using query oriented server

from other such threads by operating system mechanisms). Each such thread executes an instance of the pseudo-code shown in figure 3, which is a simple variant of that shown in figure 1.

So each boid gets the tuples about boids in its neighbourhood, delegating the notion of what "its neighbourhood" means to the environment itself. As far as the boid is concerned it is querying for "all the boids". The environment knows where each boid is in the entire world and answers a *relative* neighbourhood of the querier: the positions on the boids in the returned neighbourhood are expressed relative to that of the querier. Furthermore, only the boids that are in what the environment deems to be "the neighbourhood" of the querier are supplied in the neighbourhood.

The boid uses the returned neighbourhood information to implement the three rules of Reynolds' algorithm, using the relative positions provided in the neighbourhood, to calculate its acceleration, which is applied to its internal state, here just the boid's velocity. The environment is then updated with its external state which in this simple example is the same velocity. There is no "position" in this state, because the boid is just where the boid is. It is up to the environment to know where the boid actually is in world, which it can calculate from the boid's velocity.

Nowhere in this pseudo-code, or in the Java code actually written, is there anything about directly coordinating the activities of separate boids. All of these details are delegated to the environment, which embodies both a knowledge of the world as a whole, for example it knows that it is a toroidal space, and of the perception of the boids, that is it knows how far away a boid has to be to be deemed "not in the neighbourhood". In this simple example, it is not necessary for the environment to support a time server, as each boid agent performs the same amount of processing to update its state.

A consequence of this lack of interaction between boids is that other versions of the same code, ones where multiple boid agents are supported by each thread, can been written. Each thread sequentially executes the same code for each of the boids in its control. The behaviour of this variant is essentially identical to the thread per boid version, although requiring fewer threads.

The implementation of the environment is carried out by using a façade object [22], in the boids process, that provides a layer above the TSpaces server itself. This façade, in the TSpaces code, retrieves all of the boids from the server and filters them for locality before presentation to the querier as the querier's neighbourhood. It must be done this way because the TSpaces query mechanisms are limited to essentially pattern matching between a template tuple and the tuples in the server's tuplespace.

There are TSpaces mechanisms that could be used to implement an "interrupt oriented" server but these have, as yet, not been investigated.

6.2 RDBMS prototype

A second prototype has also been constructed that uses an RDBMS, specifically MySQL [14]. This prototype also functions well.

The code executed by the RDBMS version is much the same as for the TSpaces variant. The difference, though, is in the environment façade. The RDBMS version can be much simpler, as the process of filtering for local boids may be done using SQL in the database query itself.

7 Future work

The architecture as described is the essential core of the environment oriented approach to complex systems simulation. Future work will concentrate on two main issues.

The first issue is that of the appropriateness, or otherwise, of the two architectural patterns, query orientation and subscription orientation. This will be investigated by producing further prototype implementations that use each style, and combinations of the two.

The other issue is of more theoretical interest. When an agent makes a query (which is logically the same as describing a topic on which it will receive tuples in the subscription oriented architecture) then, as has been described, the response essentially carries with it the topology of the space in which the response is embedded. Realistic complex systems are likely to either:

1. make multiple queries each of which generates a response in a different space or
2. receive responses to a single query with varying topologies (such as near and distant birds in a bird flocking example)

Future work will look at the issues relating to how the responses in different topologies are combined, if that is feasible, and what that implies for more complicated complex systems which more closely represent the details of the real world.

8 Conclusions

The agents in real world complex systems do not directly interact via some "action at a distance"; they interact through the mechanisms mediated by a complicated environment in which they are all embedded. Producing complex systems simulations in an environment oriented manner uses environment implementations in which many complicated functions of the agents are embedded. The use of the environment oriented approach to complex systems simulation promises to raise the level of abstraction in simulation development. This approach allows design to avoid many details related to deadlock and communication. Other issues become apparent, such as how to handle the varying resolution and accuracy inherent in a typical complex real world situation. These issues can potentially be represented as a set of topologies in which real work values are embedded.

8.1 Acknowledgements

The work described here is part of the CoSMoS[2] project, funded by EPSRC grant EP/E053505/1 and a Microsoft Research Europe PhD studentship.

References

1. P. Andrews, A. Sampson, J. Bjørndalen, S. Stepney, J. Timmis, D. Warren, and P. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In *Artificial Life XI*, pages 17–24. MIT Press, 2008.
2. Fred R. M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier synchronisation for occam-pi. In Hamid R. Arabnia, editor, *PDPTA*, pages 173–179. CSREA Press, 2005.
3. J. L. Deneubourg and S. Goss. Collective patterns and decision-making. *Ethology, Ecology & Evolution*, 1:295–311, 1989.
4. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, May 2002.
5. Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.
6. David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.
7. Jean-Louis Giavitto and Olivier Michel. Data structure as topological spaces. In *Proceedings of the 3rd International Conference on Unconventional Models of Computation*, pages 137–150, 2002.
8. IBM. The TSpaces vision. http://www.almaden.ibm.com/cs/TSpaces/html/Vision.html, accessed on 6th May, 2009.
9. ISO. *ISO/IEC 9075-1:1999: Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*. 1999.
10. ISO. *ISO/IEC 9075-2:1999: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*. 1999.
11. Jini. The community resource for Jini technology. http://www.jini.org, accessed on 6th May, 2009.

[2] http://www.cosmos-research.org

12. J. M. R. Martin and P. H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4), 1997.
13. Robin Milner. *The Space and Motion of Communicating Agents*. CUP, 2009.
14. MySQL. Open source database. http://www.mysql.com, accessed on 6th May, 2009.
15. Sun Developer Network. Java Message Service (JMS). http://java.sun.com/products/jms/, accessed on 6th May, 2009.
16. Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Sci. Comput. Program.*, 41(3):277–294, 2001.
17. Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.
18. Masatoshi Seki. dRuby and Rinda: Implementation and Application of Distributed Ruby and its Parallel Coordination Mechanism. *International Journal of Parallel Programming*, 37(1):37–57, 2009.
19. Susan Stepney. Embodiment. In Darren Flower and Jon Timmis, editors, *In Silico Immunology*, chapter 12, pages 265–288. Springer, 2007.
20. Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders, editors, *25 Years Communicating Sequential Processes*, volume 3525 of *LNCS*, pages 175–210. Springer, 2004.
21. Wikipedia. Client server architecture. http://en.wikipedia.org/wiki/Client-server, accessed on 18th June, 2009.
22. Wikipedia. Facade pattern. http://en.wikipedia.org/wiki/Facade pattern, accessed on 6th May, 2009.
23. Wikipedia. Message oriented middleware. http://en.wikipedia.org/wiki/Message Oriented Middleware, accessed on 6th May, 2009.
24. Wikipedia. Multitier architecture. http://en.wikipedia.org/wiki/Multitier architecture, accessed on 18th June, 2009.
25. Wikipedia. Publish/subscribe. http://en.wikipedia.org/wiki/Publish/subscribe, accessed on 6th May, 2009.
26. Wikipedia. Transaction processing. http://en.wikipedia.org/wiki/Transaction processing, accessed on 18th June, 2009.