An Application of CoSMoS Design Methods to Pedestrian Simulation

Sarah CLAYTON 1, Neil URQUHART and Jon KERRIDGE

School of Computing, Edinburgh Napier University, Edinburgh, EH10 5DT.

Abstract. In this paper, we discuss the implementation of a simple pedestrian simulation that uses a multi agent based design pattern developed by the CoSMoS research group. Given the nature of Multi Agent Systems (MAS), parallel processing techniques are inevitably used in their implementation. Most of these approaches rely on conventional parallel programming techniques, such as threads, Message Passing Interface (MPI) and Remote Method Invocation (RMI). The CoSMoS design patterns are founded on the use of Communicating Sequential Processes (CSP), a parallel computing paradigm that emphasises a process oriented rather than object oriented programming perspective.

Keywords. JCSP, CoSMoS design methods, pedestrian simulation, multi-agent systems

1. Introduction

The realistic simulation of pedestrian movement is a challenging problem, a large number of individual pedestrians may be included in the simulation. Rapid decisions must be made about the trajectory of each pedestrian in relation to the environment and other pedestrians in the vicinity. Parallel processing has allowed such simulations to include tens of thousands of pedestrian processes travelling across large areas. The multi-agent systems paradigm has recently been used to significant effect within pedestrian simulation, typically each pedestrian within the simulation equates to a specific agent. Examples of these are described in [1,2]. In [2], the parallel aspect of the simulation is implemented using MPI.

The Complex Systems Modelling and Simulation infrastructure (CoSMoS) research project [3], a successor to the TUNA [4] research project, is an attempt to develop reusable engineering techniques that can be applied across a whole range of MAS and simulations. Examples include three dimensional simulations of blood clot formation in blood vessels, involving many millions of processes running across a number of networked computers [5,6]. The aims of the CoSMoS project are to provide a ‘massively-concurrent and distributed’ [7] infrastructure based on a process-oriented programming model. This process-oriented paradigm derives from the process algebras of CSP [8] and \( \pi \)-calculus [9]. The purpose of these methods is the elimination of perennial problems in concurrency arising from conventional parallel techniques, such as threads and locks. These problems include deadlock, livelock and race hazards.

Initial demonstrations produced by the CoSMoS research group are implemented using occam-\( \pi \) [10], a language developed to enable the direct implementation of concurrent systems complying with the principles of CSP and \( \pi \)-calculus. To this end, the Kent Retargetable occam Compiler (KRoc) [11] was developed at the University of Kent Computing Labora-

---

1 Corresponding Author: Sarah Clayton, School of Computing, Edinburgh Napier University, Merchiston Campus, Edinburgh EH10 5DT, UK. Tel: +44 131 455 2477. E-mail: s.clayton@napier.ac.uk.
tory. Similar capabilities are also provided by the Java Communicating Sequential Processes (JCSP) [12] packages, also developed at the University of Kent. As a library for the mainstream Java language, JCSP is more accessible to general programmers. It provides a means of implementing the semantics of CSP, that uses the underlying Java threading model, without the developer needing to be concerned with the details. The work described here has been created using JCSP.

2. CSP and the $\pi$-calculus

In this section, we give a brief summary of the main concepts within CSP and the $\pi$-calculus. They are part of a rich set of semantics for concurrent programming, made directly available to programmers through the occam-$\pi$ programming language and the JCSP library for Java.

CSP and the $\pi$-calculus are formalisms for designing and reasoning about concurrent systems. Development environments that allow the application of these formalisms encourage a process-oriented, rather than object-oriented approach, to programming.

2.1. Processes

Processes have their own thread of control and entirely encapsulate all their data and maintain their own state. These cannot be altered by other processes [13]. Other processes may send a message to a process requesting that its state be updated; such messages may be refused by the receiving process until it is in a correct state to deal with it. This gives two important wins for concurrent design: firstly, state never changes without the process owning that state making the change and secondly, a process never needs to manage data for requests with which it cannot deal. No locks are needed. Of course, care must be taken to avoid deadlocks caused by the refusal of messages.

2.2. Networks, Channels and Barriers

Processes combine in parallel to form networks. Processes interact through synchronising on shared events. Instead of being propagated by listeners, events in process oriented systems are built upon channels and barriers. A network of processes is itself a process, so layered architectures – reflecting the layered structures of real life – are simply modelled.

Barriers are a fundamental part of Bulk Synchronous Processing (BSP) [14]. They are directly modelled by multiway events in in CSP. They allow multiple and heterogeneous processes to synchronise their activities, and enforce lockstep parallelism between them. When a process synchronises on a Barrier, it waits until all other processes enrolled on the Barrier have also synchronised before continuing processing.

Processes communicate privately to each other using channels. Channels are synchronous. Any process writing to a channel will block until the reader is able to process the message. Although it is possible to implement buffering in these channels (JCSP supports this directly), no buffered channels are used here. One2OneChannels allow point to point communication between processes, and Any2OneChannels allow many processes to communicate with a single process. Other channel types are available but are not discussed here.

3. Implementation

The structure of the simulation is based partly on the server structure described in [2] and the description of space described by Andrews et al. in [7]. The simulation uses fine grained parallelism, and is scalable. It has three main elements, Agents, Sites, and an overall SiteServer. These are implemented as processes rather than passive objects, and communi-
cate with each other using channels, rather than through method calls. This allows processes and their data to be completely encapsulated. Any change of state or control information is communicated, as necessary, to other processes through channels.

The space to be simulated is modelled by multiple Site processes, in a way similar to [7]. The SiteServer process acts as a server to all its client Sites. A Site process acts as a server to an ever-changing set of mobile agent clients – see Figure 1. The use of a client-server architecture here eliminates the dangers of deadlock [15].

![Diagram of server layers based on Quinn et al. [2]](image)

**Figure 1.** Server layers based on Quinn et al. [2]

Agents are mobile across sites: they may deregister themselves from any given Site and migrate to another. Each Site has a register channel for this purpose, where it receives registration requests from Agents, in the form of their server channel ends. This allows communication to be immediately established between the Agent as client and the Site as server. The process of registration (by a SiteServer) is described in the code in Listing 1:

```java
private void register() {
    boolean polling = true;
    timer.setAlarm(timer.read() + timeout);
    Vector<ServerChannelEnd> servers = new Vector<ServerChannelEnd>();
    while (polling) {
        int index = alternative.select(); // wait for a timeout or channel register
        if (index == TIMER)
            polling = false;
        else if (index == CHAN) {
            ServerChannelEnd s = (ServerChannelEnd) register.read();
            servers.add(s);
            timer.setAlarm(timer.read() + timeout);
        }
    }
    for (ServerChannelEnd newserver : servers) {
        newserver.read();
        newserver.write("Hello");
        server.add(newserver);
    }
}
```

Listing 1: Code for client registration
The SiteServer operates in a manner conceptually similar to that described in [2]. Agents communicate their current location to the Site with which they are currently registered. Sites then engage in a client-server communication with the SiteServer, which aggregates all this information. In the next phase of the communication, the SiteServer returns this global information to each Site, which then passes it on to each Agent. Agents then act on this information and alter their current position. This is described in Table 1 below, and compares the three main processes of the simulation. The implementation of the pedestrian simulation is illustrated in Fig. 1.

3.1. Discover and Modify

In order to ensure that all processes are updated and modified in parallel, two Barriers are used: discover and modify. During the discover phase, all Sites are updated by the SiteServer with the global coordinates of every Agent. Each Site then updates all Agents that are registered with it.

As explained in [16,6], autonomous software agents perceive their environment and then act on it. This creates a two phase process for each step of the simulation, discovery and modification, that all processes comply with. These phases are enforced by barriers, described above. The tasks carried out by each type of process for each step of the simulation are described in the Table 1.

<table>
<thead>
<tr>
<th>Table 1. Processing Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>Synchronise on discover barrier</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Request update → Receive requests</td>
</tr>
<tr>
<td>Receive update ← Send global coordinates</td>
</tr>
<tr>
<td>Modify state</td>
</tr>
<tr>
<td>Send state → Receive state</td>
</tr>
<tr>
<td>Receive ACK ← Send ACK</td>
</tr>
<tr>
<td>Send updates → Receive updates</td>
</tr>
<tr>
<td>Receive ACK ← Send ACK</td>
</tr>
<tr>
<td>Aggregate updates into global coordinates</td>
</tr>
</tbody>
</table>

All communications between processes are on a client-server basis. In effect, a client-server relationship involves the client sending a request to the server, to which the server is guaranteed to respond [17,15]. Processes at the same level do not communicate directly with each other, only with the process at the next level up. As stated in [6]: “such communication patterns have been proven to be deadlock free”. See [15] for a proof.
3.2. Description of Space

The division of space between sites allows for a simulation that is scalable and robust, separating out the management of agents between many processes. The Site processes themselves have no knowledge of how they are situated. Each Agent class has a Map object that provides information about the area that a Site is associated with and the means with which the Agent can register with this Site. In this way, Site processes can be associated with spaces of any shape or size. These spaces can range from triangles, simple co-planar two dimensional areas, complex three dimensional shapes, to higher dimensions with dynamically forming and shifting worm-holes.

At the edges of each space, an Agent may either migrate to the next Site, or encounter a hard boundary, requiring it to change direction. This is determined by the existence of a reference to the adjacent Site, if one exists. This is a reference to the Site's register channel, an Any2OneChannel, which allows many writers (the Agents seeking to register) and only one reader (the destination Site).

The register process happens in two phases. First the Agent must inform its current Site that it wishes to deregister, during the discovery phase. During the modify phase, before any other operation or communication is carried out, the Agent writes a reference to its communication channels to the register channel of the new Site, and waits for an acknowledgement. In this way, while an arbitrary number of Agents may wish to migrate from Site to Site at any one time, these attempts will always succeed. An image from the software is shown in Fig. 2 below.

![Figure 2. Pedestrian agents in the application showing the arc of their field of vision](image)

The current work implements simple reactive agents. These contain little in the way of intelligence in making their choices. Their field of view replicates that of humans. The span of human vision is 160 degrees. Central vision only occupies 60 degrees of this, with peripheral vision on each side occupying 50 degrees [18]. The minimum distance between agents is delimited by the inner arc of their field of view. Should any other Agent approach this, they will react by choosing a different direction.
4. Results

A number of test runs were performed to evaluate how the simulation performed when the number of agents was incremented. This was done in order to demonstrate the scalability of the system. This test was carried out with one Site object, and the number of Agents incremented by ten for each run. The results are summarised in Table 2.

<table>
<thead>
<tr>
<th>Number of agents</th>
<th>Total time</th>
<th>Avg time per step (ms)</th>
<th>Avg time per Agent (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>151</td>
<td>15.11</td>
<td>1.51</td>
</tr>
<tr>
<td>20</td>
<td>412</td>
<td>20.62</td>
<td>1.03</td>
</tr>
<tr>
<td>30</td>
<td>845</td>
<td>28.17</td>
<td>0.94</td>
</tr>
<tr>
<td>40</td>
<td>1394</td>
<td>34.86</td>
<td>0.87</td>
</tr>
<tr>
<td>50</td>
<td>2057</td>
<td>41.13</td>
<td>0.82</td>
</tr>
<tr>
<td>60</td>
<td>2853</td>
<td>47.55</td>
<td>0.79</td>
</tr>
<tr>
<td>70</td>
<td>3741</td>
<td>53.44</td>
<td>0.76</td>
</tr>
<tr>
<td>80</td>
<td>5035</td>
<td>62.94</td>
<td>0.79</td>
</tr>
<tr>
<td>90</td>
<td>6207</td>
<td>68.97</td>
<td>0.77</td>
</tr>
<tr>
<td>100</td>
<td>7817</td>
<td>78.17</td>
<td>0.78</td>
</tr>
<tr>
<td>110</td>
<td>9401</td>
<td>85.46</td>
<td>0.78</td>
</tr>
<tr>
<td>120</td>
<td>11111</td>
<td>92.59</td>
<td>0.77</td>
</tr>
<tr>
<td>130</td>
<td>12923</td>
<td>99.41</td>
<td>0.76</td>
</tr>
<tr>
<td>140</td>
<td>15243</td>
<td>108.88</td>
<td>0.78</td>
</tr>
<tr>
<td>150</td>
<td>17454</td>
<td>116.36</td>
<td>0.78</td>
</tr>
<tr>
<td>160</td>
<td>20030</td>
<td>125.19</td>
<td>0.78</td>
</tr>
<tr>
<td>170</td>
<td>22251</td>
<td>130.89</td>
<td>0.77</td>
</tr>
<tr>
<td>180</td>
<td>25443</td>
<td>141.35</td>
<td>0.79</td>
</tr>
<tr>
<td>190</td>
<td>28694</td>
<td>151.02</td>
<td>0.79</td>
</tr>
<tr>
<td>200</td>
<td>31104</td>
<td>155.52</td>
<td>0.78</td>
</tr>
</tbody>
</table>

As can be seen from Table 2, the time to update each Agent during each step of the simulation is more or less constant. This is illustrated in Fig. 3 below.

These average times tend to decrease as the number of Agents increase. This reflects the overhead of setting up support processes, such as the display processes. Thereafter, the average times per Agent tend to settle at around 0.78 ms. However, beyond a certain point, the Java Virtual Machine (JVM) is no longer able to allocate any more threads and throws an exception. At the same time, as discussed below, it is unlikely that the number of Agents will exceed thirty in this application.

5. Conclusion

In this paper, the application of CoSMoS design patterns in the development of MAS simulations has been discussed. The principles of concurrent processing using non-conventional techniques based on CSP and π-calculus have been explained. The client-server pattern that guarantees livelock and deadlock free concurrency has also been discussed. This offers a firm foundation for future work, using MAS, to simulate pedestrian behaviour.
6. Future Work

Although there is an upper limit to the number of threads the JVM can allocate, it would be possible to increase the number of Agents by distributing the application across a number of JVMs and networked computers. This can be done using the NetBarrier feature of the development version of JCSP. However, this would require a redesign of the structure of the application more in line with the occoids example described by Andrews et al. in [7] than the layered server based structure of Quinn et al.'s work in [2].

Although many simple agents have been used to simulate emergent behaviour [19], the purpose of the TRAMP project [20] is the simulation of human behaviour derived from data collected by infra-red sensors. As shown in Fig. 4 actual human movements, when navigating across a space, are described by elegant and coherent curves. This is difficult to replicate using simple agents. In order to achieve this aim, agents trained using Learning Classifier Systems (LCS) [21] will be developed, and their interactions studied. The training data will allow the creation of agents that display realistic behaviours.

The aim of the TRAMP research project is to simulate simple interactions between individuals, such as overtaking, group behaviour and obstacle avoidance, at the microscopic level. Issues of emergence, at the macroscopic level, are not dealt with. The environment being studied is relatively small, a straight corridor measuring 15 × 4 metres. The density of pedestrians passing through rarely exceeds 30 people. However, this provides a rich set of data on microscopic behaviours.

References


