Sim94 – A concurrent simulator for plan-driven troops

Editors
Björn Carlson and Samuel Tronje

Computing Science Department, Uppsala University
Box 311, S–751 05 Uppsala, Sweden
+46–18–18 25 00

Project managers:
Björn Carlson and Samuel Tronje
e-mail: {bjorn, sammy}@csd.uu.se

Martin Argenius
Calle Bergman
Daniel Blomqvist
Arne Boråv
Alexander Bottema
Hugo Calendar
Massih Enayatollah
Anna Erikson
Pekka Hedqvist
Gustaf Holmkvist
Key Hyckenberg
Magnus Ingelbo

Anders Johansson
Per Jonsson
Niklas Kaltea
Martin Klint
Anders Lindgren
Terje Lundin
Pär Mattsson
Nils Meinhard
Greger Ottosson
Olle Pellijeff
Kristina Sirhuber
Johan Vestermark
Martin Wikborg
Herman Ågren
Peter Öhman
Abstract

We present the simulation system for plan-driven troops developed at the Computing Science Department, Uppsala University, 1994.

This work was performed by the members of Projekt94 in cooperation with CelsiusTech IT AB.

Traditionally, military leadership training have consisted of operations involving much personnel. This is expensive and therefore simulation is needed.

We would like the units in military leadership training to be computer controlled. To provide such units we use agent-based simulation. An agent is an entity, in this case representing a unit, that can make decisions and act accordingly. Anything from simple finite automata to sophisticated AI methods can be used for decision making. The agents may interact with other agents, the user and/or the environment. Complex behaviour can emerge due to the interaction even with very simple agents, consider for example Conways Game of Life.

The problem is to implement a simulator for military leadership training of battalion chiefs. The active entities in the simulation represents troops of 150 infantry soldiers.

The simulator system is based on the client-server model, where the server operates independently, i.e. the simulation can go on without any connected clients. The clients can connect to the simulation server to enable inspection and manipulation of the simulation.

The simulation is agent-based, an agent represents a troop and is implemented as an autonomous process in the ERLANG programming language. The agents are controlled by a set of rules and placed in a dynamic terrain where they interact with friendly and hostile troops/agents.

The digital terrain is dynamic, it can be manipulated during the simulation by creation and destruction of forbidden areas. Agents are unable to move into forbidden areas which represents blown bridges, mine fields, attacks with chemical weapons and other areas which agents should not enter. The forbidden areas are controlled by the superuser.

The simulator can import geographical data from common GIS formats e.g. ARC/INFO.
Contents

1 Introduction .......................... 1
  1.1 Introduction .......................... 1
  1.1.1 Background .......................... 1
  1.1.2 Our approach ......................... 2
  1.2 Problem ............................. 2
  1.2.1 The simulator ......................... 3
  1.2.2 Clients ............................ 3
  1.2.3 Terrain and time ..................... 3
  1.3 ERLANG .............................. 4
  1.3.1 Programming in ERLANG .............. 4
  1.3.2 The ERLANG system .................. 4
  1.4 Design .............................. 5
  1.4.1 Optimisations ....................... 5
  1.4.2 System structure ..................... 5

2 Agents and Nationality ................. 9
  2.1 Introduction ......................... 9
  2.2 Agents .............................. 10
    2.2.1 Marching Through the Terrain ....... 11
    2.2.2 Following Orders ................... 11
    2.2.3 Discrete Time Increments .......... 11
    2.2.4 Battle ............................ 11
    2.2.5 The Status of an Agent .......... 12
  2.3 The State Machine ................... 12
    2.3.1 The Different States ............... 12
    2.3.2 State Transitions ................... 14
    2.3.3 Events as a Way of Changing States 15
    2.3.4 World Events and Confirm Attack ... 16
  2.4 Possible Improvements to the State Machine 16
  2.5 Orders .............................. 17
  2.6 Nationality .......................... 18
    2.6.1 Detected and Undetected Enemy Agents 18
    2.6.2 Data Flow .......................... 18
    2.6.3 Global Reconnaissance ............... 19
    2.6.4 Nationality as a View Provider .... 19
  2.7 Implementation Outline .............. 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.1 Code Structure</td>
<td>19</td>
</tr>
<tr>
<td>2.7.2 Integration with Other Modules</td>
<td>19</td>
</tr>
<tr>
<td>3 World</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>21</td>
</tr>
<tr>
<td>3.2 The World Module</td>
<td>22</td>
</tr>
<tr>
<td>3.2.1 Synchronisation</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Battle Judgement</td>
<td>25</td>
</tr>
<tr>
<td>3.2.3 Enemy Detection</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4 World as a View Provider</td>
<td>29</td>
</tr>
<tr>
<td>4 Terrain</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Terrain representation</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1 T5</td>
<td>31</td>
</tr>
<tr>
<td>4.2.2 ERLANG representation</td>
<td>31</td>
</tr>
<tr>
<td>4.3 Integration in the simulator</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1 Logging</td>
<td>34</td>
</tr>
<tr>
<td>4.3.2 Queries</td>
<td>34</td>
</tr>
<tr>
<td>4.3.3 Changes in the terrain</td>
<td>34</td>
</tr>
<tr>
<td>4.4 Implementation outline</td>
<td>35</td>
</tr>
<tr>
<td>4.4.1 GRASS</td>
<td>35</td>
</tr>
<tr>
<td>4.4.2 The Implementation of conversion</td>
<td>36</td>
</tr>
<tr>
<td>4.4.3 Road layout</td>
<td>40</td>
</tr>
<tr>
<td>4.4.4 Water layout</td>
<td>41</td>
</tr>
<tr>
<td>4.4.5 Lookup and changes</td>
<td>43</td>
</tr>
<tr>
<td>5 Navigator</td>
<td>44</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>44</td>
</tr>
<tr>
<td>5.2 Problem Specification</td>
<td>45</td>
</tr>
<tr>
<td>5.2.1 Map Format</td>
<td>45</td>
</tr>
<tr>
<td>5.2.2 Navigation Services</td>
<td>45</td>
</tr>
<tr>
<td>5.2.3 Reconnaissance</td>
<td>46</td>
</tr>
<tr>
<td>5.3 Solution Outline</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1 Interactive Agent Approach</td>
<td>46</td>
</tr>
<tr>
<td>5.4 Implementation</td>
<td>47</td>
</tr>
<tr>
<td>5.4.1 Control Flow</td>
<td>47</td>
</tr>
<tr>
<td>5.4.2 Tracker Algorithm</td>
<td>48</td>
</tr>
<tr>
<td>5.4.3 Post-processing</td>
<td>50</td>
</tr>
<tr>
<td>5.5 Test-bench</td>
<td>50</td>
</tr>
<tr>
<td>5.5.1 Preferences</td>
<td>50</td>
</tr>
<tr>
<td>5.5.2 Supervisor Window</td>
<td>51</td>
</tr>
<tr>
<td>5.6 System Integration</td>
<td>52</td>
</tr>
<tr>
<td>5.6.1 Distribution</td>
<td>52</td>
</tr>
<tr>
<td>5.6.2 Logging</td>
<td>52</td>
</tr>
<tr>
<td>5.7 Conclusions</td>
<td>52</td>
</tr>
<tr>
<td>5.7.1 Quality of service</td>
<td>53</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 System structure of processes ............................................. 6  
3.1 Synchronisation ................................................................. 22  
3.2 The attack confirm protocol ................................................. 27  
4.1 Example raster outile. ......................................................... 32  
4.2 This is a part of the road file used by our version of the system .... 33  
4.3 .grassrc ................................................................. 36  
4.4 The grammar for the initfile. ................................................. 36  
4.5 Example of the initfile. ....................................................... 36  
4.6 The convert file our system is running ..................................... 37  
4.7 Example of the terrain array our system is using ......................... 38  
4.8 A raster file, converted and represented in ERLANG tuples .......... 38  
4.9 An example of a vector object of the original type .................... 39  
4.10 The example vector interpreted as a road ................................ 40  
4.11 The example vector interpreted as water ................................ 42  
5.1 Moving from edge to edge ..................................................... 46  
5.2 Description of control flow ................................................... 47  
5.3 Moves considered at each step .............................................. 48  
5.4 Exit Evaluation ............................................................... 49  
5.5 Moving around water ........................................................ 49  
5.6 Description of a shortcut ..................................................... 50  
5.7 Preference window for the test-bench ..................................... 51  
5.8 Command section of test-bench window .................................... 52  
5.9 Description of symbols used in display of search ....................... 52  
5.10 A situation where this tracker will fail .................................... 53  
6.1 Relay processes ................................................................. 60  
6.2 The initial design model ....................................................... 61  
6.3 State diagram for master process .......................................... 63  
6.4 Example screen for main debugging window ................................ 68  
6.5 Example screen for viewing processes .................................... 68  
6.6 Example screen for statistics ............................................... 69  
7.1 The topology of the tokenizer-buffer communication .................... 74  
7.2 The graphical user interface ............................................... 79  
7.3 The tool-box buttons from left to right .................................... 80
7.4 If a troop reaches a forbidden area it stops. The area in its view is revealed. ............................................. 81
7.5 Here you can see how a troop find its way to the desired location. 81
7.6 The dialogue window for removing orders. ........................... 82
A.1 The design of the view interface ................................. 85
Chapter 1

Introduction

by Samuel Tronje

1.1 Introduction

This technical report describes a simulator developed by 27 students as a project in the fourth year of the masters education in computing science, Uppsala, in the fall of 1994. The project amounts to approximately six weeks of full-time work. It was performed on an experimental basis and many problems were solved during the implementation phase. The project was intentionally managed without formal system development methods. The project was managed by Björn Carlson and Samuel Tronje.

By implementing this simulator we try to answer two interesting questions:

- Is a process-based implementation efficient enough for this kind of simulation?
- Is it possible to implement such a large system in a short time by using high-level tools like the ERLANG [7] programming language?

1.1.1 Background

Traditionally, military leadership training have consisted of operations involving much personnel. This is expensive and therefore simulation is needed.

The present system TRACCS [6] is an attempt to solve some of these problems. TRACCS is a computer-aided system for training of battalion chiefs. The battalion chief is placed in his real environment, equipped with maps and radios. His troops are units controlled by operators in a control centre. The computer-aid consists of a common map background where the operators can plot routes for the marker that represents their unit. Unit movement, including fuel consumption and wear of the material, is handled by the computer. The simulation state can be saved. Radio traffic is computer controlled and allows for varying quality according to the surrounding terrain. Radio traffic is recorded on tape as a part of the logging. The outcome in battle is decided by a supervising training leader.
However, this form of training still requires many people to perform auxiliary tasks as there are no computer controlled units.

1.1.2 Our approach

We would like the units in military leadership training to be computer controlled. To provide such units we use agent-based simulation. An agent is an entity, in this case representing a unit, that can make decisions and act accordingly. Anything from simple finite automata to sophisticated AI methods can be used for decision making. The agents may interact with other agents, the user and/or the environment. Complex behaviour can emerge due to the interaction even with very simple agents, consider for example Conways Game of Life.

In the implemented system there is only one type of agent (see Chapter 2), infantry groups of approximately 150 men. The agents coexist in a dynamic terrain which is a digital representation of a geographical area. The terrain is dynamic in the sense that it can be manipulated by the training leader e.g. blow bridges.

A process-based implementation of such a system is natural as concepts communication and synchronisation in the model maps directly to the same concepts in the implementation language. Each agent is implemented as a concurrent process and interaction is handled by message passing.

The system allows for extensions due to the general model. Multi-type agent simulations are easily handled by this approach, it is merely a question of specifying communication protocols between the processes/agents. In a war-gaming simulation there can be infantry-, tank- and artillery-agents, which differ mainly in fire power and movement. There can also be agents representing officers at different levels with different decision mechanisms.

Integration with other simulators, e.g. a tank simulator communicating via a DIS (Distributed Interactive Simulation) [1] protocol, only needs a special type of agent as an advanced gateway for bidirectional information relay. Agent-processes or other auxiliary processes in the simulator can also be distributed to speed up the simulation and provide robustness.

1.2 Problem

The problem is to implement a simulator for military leadership training of battalion chiefs. The active entities in the simulation represents troops of 150 infantry soldiers.

The simulator system is based on the client-server model, where the server operates independently, i.e. the simulation can go on without any connected clients. The clients can connect to the simulation server to enable inspection and manipulation of the simulation.
1.2.1 The simulator

The simulation is agent-based, an agent represents a troop and is implemented as an autonomous process in ERLANG [7]. The agents are controlled by a set of rules and placed in a dynamic terrain where they interact with friendly and hostile troops/agents.

An agent understands three orders:

- **occupy place**, march to place and defeat enemy units;
- **defend place**, march to place, build defences and defeat attacking enemy units;
- **reorganise at place**, withdraw from battle and march.

To carry out these orders the agents perform a sequence of actions such as wait, defend, attack and march.

Route planning for movement is an important part of the agent’s intelligence, all orders may require movement. When planning a route there can be time constraints, that must be satisfied. While satisfying the time constraints and avoiding enemy units and forbidden areas the route should maximise terrain protection.

Each unit performs reconnaissance within a 2 km radius. There is also air-reconnaissance with a small chance to discover enemy units. If an enemy unit or a forbidden area is found, its location is communicated to all the friendly units.

1.2.2 Clients

Clients communicates with the server through a protocol which uses sockets and TCP/IP, no special language or machine architecture is required.

Clients can operate in **user mode** or **superuser mode**. **Superuser mode** is for the training leader and **user mode** for is for the person being trained.

In superuser mode all units are shown on a map. New units can be created and existing units can be destroyed. Forbidden areas can be created and destroyed. Information about troops can be viewed and orders can be issued to troops.

In user mode only friendly troops and discovered enemy troops are displayed on the map. Information about friendly troops can be viewed and orders can be issued to friendly troops.

1.2.3 Terrain and time

The digital terrain is dynamic, it can be manipulated during the simulation by creation and destruction of forbidden areas. Agents are unable to move into forbidden areas which represents blown bridges, mine fields, attacks with chemical weapons and other areas which agents should not enter. The forbidden areas are controlled by the superuser.

The simulator can import geographical data from common GIS formats e.g. ARC/INFO.
The simulation is fair i.e. agents with the same fatigue are able to move the same distance in the same time. This introduces the problem of time.

Time is discrete, divided in consecutive intervals. During an interval the agents autonomously simulate the actions they will carry out in the amount of time the interval represents. At the end of the interval each agent communicates its new state to the other agents.

1.3 ERLANG

The simulator is implemented in ERLANG.

ERLANG [7] is a concurrent functional language with support for distribution. Process communication is obtained by asynchronous message passing.

1.3.1 Programming in ERLANG

ERLANG has

- a module system, all functions belongs to a module;
- no global data, only exported functions are visible outside a module;
- no primitives for iteration, recursion is used instead;
- two ways of forming compound data types: lists and tuples;
- single assignment variables which are assigned values by the matching primitive ‘=’. This primitive performs matching as opposed to unification used in e.g. Prolog.

A function is defined by a sequence of rewrite rules. When a function is applied a search mechanism examines the rules in turn and commits to the first rule with a matching head.

The `spawn/3` built-in function is used to start a new process. The '!' primitive is used to send a message to a specific process and the `receive` primitive is for receiving messages.

1.3.2 The ERLANG system

The ERLANG system runs as a single process under the host operating system. In the following, `node` will be used to denote a running ERLANG system and `process` will be used to denote an internal ERLANG process.

ERLANG has its own internal process scheduling. Large parts of the runtime environment is implemented in ERLANG running as processes in the ERLANG operating system. There are several implementations of ERLANG including both byte-code emulators and compilation to C.

The communication with external programs is handled conveniently by the `open_port/2` primitive, if speed is extremely important there is also a possibility to link-in drivers in the system.
Distributed programming is integrated in ERLANG. Processes can be started on remote nodes and messages can be sent using the same primitive '!' as for processes on the same node. This is done transparently even if there are nodes running on different architectures. Messages are guaranteed to be delivered.

### 1.4 Design

Some optimisations that affected the design will be presented next, followed by an overview of the system.

#### 1.4.1 Optimisations

The agents need to know each others positions. If each agent has to send a message to all the other agents about its new position at the end of each time interval, for \( n \) agents, \( n \times (n - 1) \) messages will be sent each interval to update positions.

The *world* process is an optimisation to bring down the number of messages for position updates. It keeps track of the agents positions and at the beginning of each interval the *world* process calculates which agents are in each others vicinity, sends a message with the result to each agent and at the end of the interval collects the new positions. This optimisation brings down the number of messages to \( 2n \).

Synchronisation is naturally a task for the *world* process. From the agents point of view an interval starts with a message from the *world* process containing identifiers of the agents in the vicinity and ends with sending a message to *world* containing the new position of the agent.

The *world* process is also responsible for judging the outcome in a battle situation.

Information about discovered enemies and forbidden areas is communicated to the friendly agents which raises the same communication problem as for position updates. The *nationality* processes is an optimisation to handle this information. There is one *nationality* process for each force to which the agents report their discoveries.

The problem of planning a route from point A to point B is computationally expensive. The *navigator* processes, at least one for each force, are an optimisation which makes it easy to distribute the computationally most expensive part of the simulator and to handle the different views of the terrain of the different forces.

#### 1.4.2 System structure

The overall system structure will be presented in this section.
Figure 1.1: System structure of processes
Agents

The agents are controlled by a state machine with a memory. The memory is necessary to keep track of strength, fatigue, position and other parameters.

A message from the world process containing enemy agents within the reconnaissance radius marks the start of a time interval. After that the agent first decides what to do. Second, it takes appropriate action, simulating an amount of time equivalent to the length of the interval, to achieve the goals decided upon. Third, it reports the new state, including the new position, to the world process. Finally, it waits for the interval.

The actions may involve communication with the nationality processes to inform the friendly agents about enemies and forbidden areas. If the agent is moving there is also communication with the terrain process to check that no forbidden areas are entered. If the agent needs movement route planning the terrain process is consulted.

World

The world process holds a database with records for all agents. This database is used to calculate which agents are in each others vicinity. The interesting parts of the result is sent to the agents which marks the start of a new interval. Then the world process collects information from the agents to update the database.

All agents wait until the interval starts and the world process wait until all agents have finished their actions before starting a new interval. This synchronises the agents. If the simulation is paused the world process simply refrain from starting a new interval.

If there is an agent attacking another agent, the world process takes care of battle judgement to determine the losses.

Terrain

The geographical data is pre-processed to an internal format with the GIS-system GRASS.

The terrain is a $25 \times 25$ km area divided in $100 \times 100$ m squares. Water is approximated only on the edges delimiting the squares. Roads are approximated between the centers of adjacent squares crossing the edges in a straight angle.

The representation of a square contains a terrain type and information about which edges are water and which are crossed by a road. A water edge can only be crossed if there is a road on the same edge. There is no altitude information.

The terrain is represented as a 250 element tuple of tuples where the inner tuples are 250 element tuples of squares. This is a common representation of a two-dimensional array in Erlang.

When the agents move along their routes, the terrain process is frequently queried about the geography along the route.
Route planning requests from the agents are relayed to the **navigator** processes through the **terrain** process.

**Navigator**

The **navigator** processes plan routes from point A to point B given certain constraints. There are time constraints of the form *it may not take more than three hours to go from A to B*. There are also geometric constraints of the form *I want to go no closer to C than 3 km while going from A to B*.

The **navigator** processes use agent based planning. A search-agent starts at point A and tries to find a safe way to point B satisfying the constraints. The search is guided in B’s direction. In each square a local choice is made of where to go next. If the choice is too hard the search-agent may split in two to explore different alternatives.

If a path is found that satisfy the constraints it is returned to the querying agent, otherwise a “no path found” answer is returned.

**Master**

The **master** process controls the simulator, creates and destroys processes. If the simulator is distributed the **master** process is in charge of the distribution.

The **master** process allocates a UNIX port for client connections and handles the communication with the clients via a socket.

Logging of the system status for robustness is done frequently. A saved status can be reloaded and the system restarted.

**Clients and tokenizers**

The clients are implemented for X windows using Scheme Tk (STk). STk is a scheme interpreter integrated with the GUI tool kit Tk, usually used together with TCL (Tool Command Language). As part of the project the socket library was interfaced with STk for communication with the simulator.

The **tokenizer** processes are the internal representation of the clients in the simulator. A **tokenizer** process receives messages from a client. These messages can be requests to e.g. create a new agent, destroy an agent, get information about an agent and issue orders.

A **tokenizer** process has a view of the simulation, in terms of agents and forbidden areas, given by the **world** process for the superuser and a **nationality** process for normal users.

The clients subscribe to events through their **tokenizer** process. Events are e.g. new position for an agent, discovery of a forbidden area and creation of a new agent. The **tokenizer** process sends a subscription request to the process where the event will occur. When the event occurs the process sends a message to the **tokenizer** process which relays the message to the client. This simplifies the clients task as there is no need to constantly ask for information.
Chapter 2

Agents and Nationality

by Hugo Calendar, Massih Enayatollah, Anders Lindgren, and Johan Vestermark

2.1 Introduction

Agents is a state-based implementation of autonomous interactive military units, written in the Erlang programming language.

Each agent represents a troop of men, which moves through different types of terrain or on roads. The speed at which an agent can move, the percentage chance that an agent detects an enemy agent, and the amount of protection from enemy fire all depend upon the type of terrain the agent in question is in or on. The agents exist in one or more teams, which are hostile to each other.

The agents in the simulation are divided into teams, and the agents on each team cooperate to overcome the other teams. The simulation is steered by User clients, (see chapter 7), which is to say military trainees, who give the agents orders. The simulation has no built in artificial intelligence to date, and is driven by a simple state machine, in conjunction with the orders that the User client gives. Thus the way the simulation plays out is largely up to the user. The simulation is a simple, yet sophisticated, machine, which shows its users what happens in the situations it is given.

All agents on a team communicate with each other, and thus if one agent on a team detects an enemy agent, the rest of the agents on its team are automatically notified of the newly detected agent’s presence.

Interaction among agents on one team is presently limited to communicating the positions of the detected enemy agents. More complicated cooperative attack patterns are not programmed into the system, and they can only occur as a result of manual guidance.

Agents on different teams interact with each other aggressively, by doing battle with them. Agents can do battle with enemy agents in their attack, defend and retreat states of action. If they don’t want to do battle, agents also avoid their enemies, as a more subtle form of interaction.
Agents can fortify an area during the simulation, thus raising their intrinsic protection when they defend the ground they’ve fortified; protection is also dependent upon the terrain type. Beyond the mentioned states in which agents can do battle, agents can also wait, rest, fortify, and march.

Since intercommunication between all of the simulated agents would be very costly to do explicitly, Nationality serves as an optimisation, making an $O(n^2)$ communications operation only $O(n)$; Nationality takes the place of intercommunication between agents on one team by collecting information from all of the agents on that team and then redistributing the compiled union of that information to its agents.

Agents operates asynchronously, and does not assume that any part of the simulation runs synchronously, even though the other modules are mostly written to run synchronously. Future rewriting of the other modules can therefore optimise performance by making the other modules run asynchronously.

Nationality provides an interface, or View, (see section A), through which a User client of the system can observe its team of agents. Put simply, each team of agents is represented by its nationality, which can have one or more interfaces.

Navigator is yet another optimisation, (see section 5), which does all navigating for all the agents, upon their request. Ideally, the path Navigator chooses for the requesting agent should be tailor made according to how much time the agent has to reach the location. The path should best shield the agent from detection, given the time constraints. At present, Agents sends Navigator timing information, which at this writing is not fully utilised, and side effects can include the failure to find a path, though the time limit might be within the limits of what is possible. (see Section 5)

Terrain keeps track of changes in the terrain, (see section 4), and reports the changes to the terrain it is in or on to the agent. World handles interaction between enemy agents by informing an agent of neighbouring enemy agents, and judging the outcome of battle.

2.2 Agents

Each agent is represented by a separate process, which performs that agent’s duties. The agent follows orders, possibly many in a sequence, as far as it is possible to follow them. The maximum number of men in a troop, and the initial number of men in a troop are set by the Superuser, (see section 7).

Agents is the heart of the simulation, and in a sense the other modules are just optimisations of what agents would otherwise do themselves. Therefore, Agents is largely given free reign to alter anything and everything in the simulation – it is up to the agent processes to behave reliably and within the constraints of the rules of the simulation, and to use the the services of the other modules appropriately.
2.2.1 Marching Through the Terrain

Agents move through a simulated terrain with the help of Navigator, (see section 5), which returns a path to an agent upon its request. A path includes a list of points to follow, and even the terrain associated with the different parts of the path. Requests are sent with information about the path’s starting point, destination and time the path is allowed to take to traverse. It is expected that Navigator provide the path with the best cover from attack, given the time constraint. Specifying a most direct path without regard to safety is also an option, and is used during attack and retreat.

Navigator also offers a function to find a path around a point, but this is not at present used by Agents.

While traversing a path, an agent updates the closer segments of its charted path every tick, and so keeps informed about the changes that may have happened in the terrain since the path was charted, (see section 4).

2.2.2 Following Orders

The agents follow orders, sent to them from User clients. Completion times can be specified for each order, and an agent’s orders are carried out (or reported impossible) in time sequential order. Three orders are possible and are (1) to take a location, (2) to defend a location, and (3) to reorganise to a location. Orders are covered more in depth in the orders section.

2.2.3 Discrete Time Increments

Simulation time is broken down into discrete time increments, the length of which is set by the Superuser. An agent performs the amount of work, (i.e. marching, waiting, etc.), appropriate for the length of the time increment, then informs World that it is ready, and waits for an indication from World that it may continue with the next time increment’s activities, (see section 3). Synchronisation with the agents’ Nationality processes also occurs at the beginning of each time increment.

2.2.4 Battle

An agent engages in battle if it is attacked, or given the ordered to attack. Except in these situations, the agents do their best to avoid each other. The only way for an agent to reach the attack state is through the take order. During an attack, the agent uses ammunition, but is unable to reload, and must thus stop from time to time, (technically, defending when stopped), to reload.

An agent must ask World to engage in battle, and, once engaged, has its status variables, (e.g. number of men, ammunition, etc.) adjusted by World. Before each tick, World asks an agent in the attack state if it wishes to continue its attack through the next time increment, and the agent answers in either the positive or the negative.
If an agent has less than 20% of its maximum manpower, the troop is dissolved, and the manpower is transferred to a neighboring troop on its team.

2.2.5 The Status of an Agent

The status of each agent is stored in a structure and is passed as the only variable through each recursive loop. The structure contains a number of items\(^1\), which can be examined, changed or deleted from the structure through the standard status item retrieval and updating functions. Certain items that are common to all agents of a specific class are not stored in the structure, but can still be retrieved in the same way that the other items are retrieved. The standard status item retrieval function can even be used to retrieve the characteristics that the agent has in a given type of terrain. In this case, the terrain type must first be inserted into the status structure.

2.3 The State Machine

The state of an agent, that is to say, what it is doing and can be doing at the next moment, can be represented by a state machine, with the modification that the current order steers what state an agent returns to at certain times. The agent process goes through one main loop, but checks the status structure on each pass, does the appropriate activity, and changes the Status structure as necessary.

A diagram of the state machine is not included in the text for the simple reason that it would be too complicated. Most of the states would have transitions to most others, and including the reasons for the transitions would make the graph unreadable. Instead of such a diagram, we include descriptions of the states, followed by descriptions of the transitions.

2.3.1 The Different States

Here we describe the different states, as in our implementation with the addition of the planned for state transitions described in the last subsection.

In states where an agent moves through the terrain, the agent receives complete paths from Terrain, but continually confirms the path at the beginning of each time increment, in case the terrain has changed since the original path was charted. For efficiency, only the first part of the path is updated. If an agent comes upon terrain that has been made unpassable, (currently only possible if the Superuser changes the terrain manually), it stops and notifies Terrain of the discovery.

\(^1\)The status structure is a tagged list of tagged items, where tagging means putting something in a 2-tuple with an atom describing what the data is. Representation as list is highly inefficient, and leaves a good deal of room for improvement, since accessing an item in the list is an \(O(n)\) operation, and the Status structure is accessed a lot. A better solution might be an ordered tree structure.
Wait

An agent in the wait state stays still but alert. This is the default state when there is no order to follow and no stimulus to react to.

Rest

An agent in the rest state is more passive than an agent in the wait state, and it can take a longer time for the agent to react to a stimulus.\(^2\)

Defend

While defending, an agent holds a position, fires at the closest enemy agent within range, and simultaneously stocks its ammunition supply. The agent can also move towards a position in the defend state, if it has an order to defend that position.

Retreat

When retreating, an agent takes the fastest path away from the attacking agent. This means that the agent must ask for a new path from Navigator.

Fortify

Fortification is the process of building up permanent protection in the terrain. The protection gained at any point only exists as long as the agent stays there. If the agent strays from the point and then returns, the fortification has to be rebuilt.\(^3\)

March

This is the most basic of the modes of movement, (the other states that Agent can move in are defend and retreat). One finds oneself in the march state as a result of a current take or reorganise order.

Attack

When attacking, an agent moves towards its nearest enemy, while firing its weapons at it. The immediately available ammunition declines steadily during attack, which is not true when defending. The agent is done when the goal has been reached and all the enemies within fire-range have withdrawn or been beaten.

\(^2\)At present there is no rest value, which might increase when resting and decrease with activity.

\(^3\)This could be changed in future implementation, if Terrain is changed to remember changes in fortification.
2.3.2 State Transitions

The following state transitions are currently implemented. The default state of an agent is the wait state.

Wait

The transitions from wait are as follows.

- If the agent is attacked or if enemies are discovered within range, the state changes to defend.
- If time passes such that an order becomes current, the state becomes that which is dictated by the order.

Rest

The transitions from rest\(^4\) are as follows.

- If an enemy is detected the state changes to wait.
- If time passes such that an order becomes current, the state becomes that which is dictated by the order.

Defend

The transitions from defend are as follows.

- If the enemy agent retreats, the state becomes fortify.
- If time passes such that an order becomes current, the state becomes that which is dictated by the order.

Retreat

Transition from retreat are not implemented.

Fortify

The transitions from fortify are as follows.

- If the agent is attacked, the new state becomes defend.
- If the agent discovers an enemy, the new state becomes wait.
- If maximum fortification is reached, the new state becomes wait.

\(^4\)Notice that rest can only be reached as a result of the reorganise order
March

The transitions from `march` are as follows.

- If the agent is attacked, the new state becomes `defend`.
- If an obstacle is encountered while walking the new state becomes `wait`.
- If the agent is attacked, the new state becomes `defend`.
- If the agent arrives at the destination, the new state becomes that specified by the current order.
- If the current order is to take a position and a point has been reached just outside the enemy's detection range, the agent stops and waits until the order is to be carried out, and then enters the `attack` state.

At present, Agents fails if it cannot reach its destination in time by the current path.

Attack

The transitions from `attack` are as follows\(^5\).

- If all the enemy agents within weapon range are conquered or withdraw out of range, new state becomes what is specified by the current order, or, if no planned activity exists, the new state becomes `fortify`.
- If the size of the troop becomes less than 75\% of the size at the beginning of battle, the new state becomes `defend`.
- If the ammunition readily available drops below 25\% of the maximum, the new state becomes `defend` for five minutes to allow for reloading, and then returns to `attack`.

2.3.3 Events as a Way of Changing States

Agents uses Events as a tool to change states. Throughout the time increment during which the agents perform their maneuvers, routines to check the different aspects of an agents' status are called, and if a state change might be warranted, the function `next_state/2` is called with an atom representing the state to go to, or, if the next state isn't clearly defined in the monitoring function, an atom representing a possible cause of a state transition is passed. From `next_state/2`, the status is altered, if appropriate, causing the calling agent to go to the appropriate next state.

\(^5\)The agent only attacks when ordered to. There is no transition to attack except through order.
Example 2.3.1 If an agent finds itself in a situation where it has to defend itself, which is to say, if either World notifies the agent that it is being attacked, or the defend order becomes current or must be resumed, then next_state(Status, defend) is called, and the necessary changes to Status are made and returned.

2.3.4 World Events and Confirm Attack

World also sends Agents events, which serve to inform about attack situations. For example, if an agent becomes attacked, World sends this information to the attacked agent at the beginning of the next time increment.

2.4 Possible Improvements to the State Machine

There are a number of improvements that might be made to the state machine as it exists now, that we can imagine. This section presents those possible improvements, and describes how they might be added to the current state machine. Many of the improvements are planned, and thus allowed for in the code structure, and many are even prepared for with already written supporting code.

Rest

An agent does not, at present, become tired at any point, but pushes on tirelessly. To make the simulation more realistic, one could associate a rest value with each agent. The agent would then become tired after some amount of activity, and crave rest, which would put them in a less aware state, as it does in real life.

For example, if an agent was in the wait state, and it's rest value became inordinately low, (10% of the maximum value, perhaps), it would automatically go to the rest state.

An agent might have a rest value of 100% at the beginning of the simulation, also possibly setable by the Superuser. The rest value could reach 0% after 24 hours of activity, and it would follow that the rest value rose $\frac{1}{24}$ of 100% for every hour of rest, and the rest value would steadily decrease at an equal but opposite rate at all times when the agent was not resting. An agent in the rest state who reached a rest value of 100% would change into the wait state.

If an agent in the rest state were attacked, the lessened awareness would cause some delay in the change of the agent’s activities. It might first go into the wait state before going through the transition to defend, or a certain amount of time might be required to pass before a direct transition to defend. Other transitions from rest would function similarly.
Defend and Retreat

Loss of men does not at present cause a transition out of the defend state. Such a transition might, however, be desirable, to simulate real life situations more accurately. For example, if a troop became less than 50% the size it was at the beginning of the simulation, it might enter the retreat state.

Also, there are at present no transition from the retreat state, which, realistically, there should be. For example, if the retreating troop gets out of firing range of the enemy it is engaged with, the new state could become wait or some other state dictated by the current order.

March

More intelligent behaviour could also be simulated when an agent is in the march state.

At present, an agent does not react if it detects an enemy agent when it is marching. It might then be a good idea, however, to determine if there is a danger of being detected if the current path is followed, and otherwise to reroute.

Also, if it at any time were determined that an agent could not reach its destination in time by the current path, a new path could be routed to allow for the successful completion of the current order.

2.5 Orders

Agents carry out orders, and remain in the wait state by default, if there are no orders to carry out, and no stimuli to cause another state to be invoked. An agent can have several orders, but only one current order. The current order is chosen by looking at the orders and determining which has the earliest finish time specified. If the destination specified by an order cannot be reached in time, the order is reported undoable, and the next appropriate order is made current by the described method.

There are three types of orders, and all orders have a destination value included in them, require movement towards this position, and have a subsequent prescribed action. An order may or may not have a prescribed time before which the order is to be carried out.

These are the possible orders:

- **take**: Go to a position and attack it [at a given time].

  The agent waits just outside the range of detection, and then attacks at the specified time. Movement towards the perimeter before the attack is done in the march state. If there is no finish time included in the order, the agent does not wait at the perimeter, but begins its attack there immediately.
• **defend**: Go to a position and defend it [at a given time].

Movement towards the destination is done in the **defend** state. Upon arrival, the state does not change.

• **reorganise**: Go to a position and rest [at a given time].

The agent uses the **march** state to move to the destination, and then goes into the **rest** state.

Orders are sent by User clients, (see section 7), to agents on its team. When an order is sent to an agent, the agent responds not only by inserting it in its status, but also by sending the client that gave the order a reference to the order, by which the order can be changed or withdrawn. There are utilities to change any of the variables in an order, return all the variables in an order, or return information about all the orders than an agent has.

### 2.6 Nationality

Nationality is in charge of collecting information from the agents, compiling it, and then redistributing it. Nationality also provides User clients with a View of its team of agents.

#### 2.6.1 Detected and Undetected Enemy Agents

The method by which agents detect enemy agents deserves some explaining. The World module, which knows the location of all agents, sends each agent a list of all agents within reconnaissance range, which the receiving agents willfully ignore, except for the purposes of determining which enemy agents they really are supposed to have detected.

Each agent randomly chooses a number of yet undetected agents to detect, so to speak. That isn’t to say that a certain number must be detected each time, or even any at all – the dice is thrown for each agent within detection range. An important factor in whether and enemy agent is detected or not is the terrain they are in or on, and the protection from detection it provides.

#### 2.6.2 Data Flow

After the agents have themselves detected enemy agents, they send their lists of detected agents to their nationality, which takes the union of the detected agents, creating the compiled detected enemy agents list. Nationality then also includes in this list those agents currently detectable which were part of the list during the last time increment. Then this list is sent back to all of the nationality’s agents, and thus the agents have information about all enemy agents which their team has detected before they start to perform each time interval.
2.6.3 Global Reconnaissance

In addition to organising and redistributing information among the agents on its team, Nationality also simulates global reconnaissance by randomly making previously unknown enemies, sent as a list from World, (see section 3), known to its team members.

2.6.4 Nationality as a View Provider

One of the responsibilities of Nationality is to make information available to the clients. This responsibility is shared with World, and is therefore done with a common View interface (see section A).

Through Nationality, clients can see how a team of agents sees the world. This isn't necessarily a correct view, as the team of agents may not have discovered some changes in the world.

2.7 Implementation Outline

The Agents and Nationality modules, as well as most other parts of the project, are implemented in the Erlang programming language, and communication between processes is done through remote procedure calls, using message passing as the communications medium.

Though many of the modules of the simulation use synchronous communication, and Agents operates asynchronously, Agents does not assume that any part of the simulation runs synchronously, allowing for future rewriting of the code to improve upon the performance by making interaction fully asynchronous.

2.7.1 Code Structure

The code is all written in the Erlang programming language, and is structured such that all activity is controlled from the main loop, called loop/1.

The actions of an agent are executed by functions with names like do_action_march and do_action_attack, but these functions only do a minimum amount of work, and then return to the main loop, mentioned above, which in turn calls the action function again until the time increment is used up.

Though communication is done through message passing, the other modules of the simulation call Agent functions which do the message passing for them, and return the desired result.

2.7.2 Integration with Other Modules

All interaction with clients is done through the View interface, (see section A).

Navigator makes available to Agents functions to find paths between two points. Time constraints can be specified, and the path safest from detection which fits the constraints is expected in return.
Terrain offers functions to see the true characteristics, or those seen by a team of agents, of the terrain of the simulated world. In particular, Terrain offers a function that updates information about a path, previously returned to an agent by Navigator, so that the agent can keep up to date on its immediate surroundings.

World synchronises the agents, by sending them “ticks” every discrete time interval, the length of which is set by the Superuser client. The agents may only perform that time increment’s amount of work, and must then report back to World, which waits until all agents have reported back before issuing a new tick.

World also judges battle damages between agents, and redistributes manpower if an agent becomes too small.
Chapter 3

World

by Calle Bergman, Niklas Kalte, Greger Ottosson, Kristina Sirhuber

3.1 Introduction

This article is devoted to the World module of the system. The task of World in the simulation is to tie the agents together in terms of time and space, and to provide a view of the world to clients.

The responsibilities of World can be divided into three groups. First there is the problem of synchronisation and the notion of time, present in any simulation system. Due to our specific simulation area and our choice of implementation language\(^1\), our solution is not entirely part of the standard framework for simulations. The activity of our simulation is not single events being triggered, but rather concurrently executing processes, forcing our solution of synchronisation to differ from e.g a discrete events simulation using a time queue.

Secondly, World handles some of the simulation-specific work for the agents, namely enemy detection and battle judgement. The reason for their existence is simple; this information and these actions are vital to an intelligent behaviour of an agent. The reason this functionality is present in World are partly practical and partly due to efficiency.

Finally, World is responsible for providing a view of the simulation to the connected clients. The information retrieved by a client from the World is the “real” one, i.e. the status of agents and facts about the terrain are correct. This opposed to information from a nationality, whose information are based on its—not always complete—knowledge.

In conclusion, World handles the following:

- synchronisation of agents at discrete time intervals,
- detection of enemy agents in an agent’s vicinity, subject to possible recognition,

\(^1\)Erlang — A functional language with declarative syntax and extensive support for concurrency, (see Section 1.3)
judgement of the outcome of battles between agents, and

- passive and active provision of information about the current state of the world.

3.2 The World Module

Here we consider the World’s part in the simulation and its possible problems and solutions.

3.2.1 Synchronisation

The usual way of programming simulation systems is to use a time queue to achieve synchronisation [10]. Every simulating agent is simulated one at a time as a discrete event. The first event in the time queue is always executed, (the time queue is sorted by time), and after it is finished it is put back into the time queue at its proper place, given which time its next
action is planned to happen. This is repeated until the time queue is empty or the simulation is explicitly stopped.

In contrast, our choice of solution is to let every simulating agent be represented as a concurrently executing process. Agent synchronisation is achieved by dividing the simulation time into discrete intervals. These intervals are separated by ticks. A tick is a message sent to all agents, informing them that a new time interval has begun. Between two consecutive ticks, no synchronisation of agents occurs.

World initiates each interval by sending a tick to every agent, (see Figure 3.1). Each interval is completed when every agent has notified World that it has completed its task for the current time interval and World has performed its own tasks.

Between each time interval, the following tasks are performed by World:

1. Calculations as to the outcome of battles are made. The outcome is that of the last simulation time interval, (see Section 3.2.2). This part is represented by Battle Judgement, (see Figure 3.1).

2. Initiation and continuation of battle, a protocol to make sure that an agent really wishes to start or continue a battle, (see Section 3.2.2). This follows right after Battle Judgement, (see Figure 3.1), and is labeled Attack confirm protocol.

3. All enemy agents within each agent’s surveillance area are found, (see Section 3.2.3). This corresponds to Enemy detection, (see Figure 3.1).

Almost all information the World propagates to agents are sent to them with the ticks. This is an optimisation done to minimise the communication with the agents.

Information that is propagated to the agents with each tick is:

- **Status** — there is a possibility that there has been some changes in an agent’s status due to battle, (see Section 3.2.2).

- **Event** — informs an agent about its status in battle, i.e. *attacking* or *defending*, (see Section 3.2.2).

- **Enemies** — all enemies within an agent’s surveillance area, (see Section 3.2.3).

- **Time** — there are two different sorts of time propagated to agents, *simulation time interval* and *the current simulated time*, see below.

Time is an essential part of any simulation. Our solution is based on three different sorts of time, as follows:

1. **Simulation time interval** — which dictates how long a time interval is from the simulated agents point of view, i.e. for how long the agents should simulate their actions.
2. **Real time interval** — this is the time agents have to perform their simulation time interval, i.e. how fast they have to execute the simulation time interval.

3. **Current Simulated time** — which indicates for how long the simulation has been active in terms of accumulated simulation time intervals.

If one is only interested in real time simulation, i.e. when there is a one to one correspondence between the flow of time for the simulating agents and the real world, then it is sufficient to use the simulation time interval. In this case the simulation time interval also takes the part of real time interval.

**Example 3.2.1** Let the simulation time interval be 5 minutes. Then each agent simulates it action for this amount of time and their simulation may only take a maximum of 5 minutes to perform.

This may not always be sufficient and we sometimes want to run the simulation faster or slower than real time. Then we need the real time interval, which allows us to configure the simulation as it pleases us.

**Example 3.2.2** If we are given a simulation time interval of 6 minutes and a real time interval of 1 minute, then we can make a 1 hour simulation in only 10 minutes. (This is useful when we have powerful computers.)

**Example 3.2.3** If we are given a simulation time interval of 1 minute and a real time interval of 6 minutes, then we can perform a 10 minutes simulation in 1 hour. (This is useful when the computers can not keep up with the load.)

The simulated time is provided to give a reference for how long the agents has been simulating.

**Synchronisation problem**

There exist some kind of problems when simulating which are not unique to concurrent simulation systems. Since we use discrete time intervals, there may arise anomalies when agents do not interfere with other agents though they really ought to. For example, two agents could have intersecting paths, but not recognise each other. This due to the fact that they follow the path during a single simulation time interval and enemy detection is only done between simulation time intervals.

The risk that this anomaly should occur can be minimised by setting the simulation time interval sufficiently small. The size of the interval is dependent of the speed and sight of the agents.
3.2.2 Battle Judgement

A central part of the simulation is the ability for agents to participate in battles. For this purpose there is a need for an objective judge, World. The battle judgement is done in a separate function executed when all agents have completed their simulation time interval ((see Figure 3.1)). The algorithms, described later, are specified in [8].

When an agent decides to attack another agent, the agent must make a request (to World) to perform the attack. When the agent makes the decision it is based on knowledge about the other agent’s old status, i.e. the other agent’s status when the current simulation time interval started. Hence the attacking agent needs a chance to confirm the request, now based on the other agent’s current status, i.e. the status when both agents have completed their simulation time interval. In addition the agents’ status may have changed due to battle judgement, which is done after each completed simulation time interval. These changes in the agents’ status may affect the agent’s decision to continue the attack or not. We have solved these problems with a message-based attack confirm protocol, which is executed after the battle judgement, (see Figure 3.1). The protocol will be explained later in this section.

To simplify the reading of the following section we introduce some terminology. First we define two kinds of requests:

Definition 3.2.1 explicit request — An agent which have decided to attack another agent must explicitly, by sending a message, ask the World for permission to attack the other agent. Such a message is an explicit request.

Definition 3.2.2 implicit request — An agent which already is in battle and in some previous simulation time interval has made an explicit request, will as a consequence of the battle judgement make an implicit request, i.e. the world will treat the agent as if he had made an explicit request. No message is sent by the agent.

In a battle we always consider one agent as attacking agent and the other as defending. The attacking agent is the agent which initiated the battle, i.e. the agent which decided to attack the other agent. Hence the following two definitions:

Definition 3.2.3 Attacking agent — An agent which in the last simulated time interval has made an explicit or implicit request.

Definition 3.2.4 Defending agent — An defending agent is an agent attacked by one or more attacking agent(s).

During the battle we keep information about which agents are attacking and defending. This information is needed for the attack confirm protocol.
The algorithms

The outcome of a battle depends on several status parameters, for example firepower, manpower, and protection. These values are provided by the agents involved in the battle. The parameters subject for changes in the agents’ status are manpower and ammunition. The changes are calculated in two different functions. The algorithm for changes in manpower, for an agent \( A \), can be expressed mathematically as:

\[
\Delta M / \Delta t = -k \times E \times F \times P
\]  

(3.1)

where \( \Delta M \) is the reduction in manpower, i.e. how many men have died, \( E \) and \( F \) are the enemy’s manpower and firepower respectively, \( P \) is \( A \)’s protection as a product of protection due to the terrain and other factors which could increase the protection. For example, \( A \) may have built a fort. \( F \) and \( P \) should be set relatively between 0.0 and 1.0. For \( F \), 0.0 means no firepower and 1.0 maximum firepower. For \( P \), it is the opposite, i.e. 0.0 means maximum protection and 1.0 no protection. \( \Delta t \) is the the length of the last simulation time interval for which the judgement is done. Finally, \( k \) is a constant which gives the ratio of well aimed shots.

**Example 3.2.4** The reduction in manpower, for an agent \( A \), with a time interval \( \Delta t \) equal to 1.

\[
\Delta M = -0.05 \times 100 \times 1.0 \times 1.0
\]

In example 3.2.4 \( \Delta M \) is \(-5\), i.e. 5 men have been killed and that amount should be removed from \( A \)’s status parameter manpower. In the example the parameters \( F \) and \( P \), are set to 1.0, which means that the enemy has maximum firepower and \( A \) has no protection (probably standing in an open field). The constant \( k \) set to 0.05 means that 5 out of 100 shots are well aimed and deadly.

In a similar way the algorithm for reduction in ammunition, for an agent \( A \), can be expressed mathematically as:

\[
\Delta A = M \times F \times \Delta t
\]  

(3.2)

where the parameters have the same meaning as in (3.1), but now they all represent the status parameters of \( A \).

The agents’ changed parameters are propagated to the involved agents with the tick in the Status parameter, (see Section 3.2.1).

An internal effect of the battle judgement, is that the attacking agents will make implicit requests to continue their battles.

Since the algorithms are represented as separate functions they could easily be modified to involve more or other parameters to give a better model of battle.
Attack confirm protocol

To avoid anomalies, e.g., an agent attacking another agent which no longer exists or has run away, we need a protocol for confirming attacks.

To make this protocol work a battle is always initiated by an attacking agent with an explicit request. In figure 3.2 agent A requests an attack on agent B. The request is later confirmed or canceled by the attacking agent.

When the battle judgement is done, all attacking agents are selected. According to definition 3.2.3 this will include all agents with explicit as well as implicit requests. For each of these agents they are asked to confirm their requests, considering possible changes in own status and enemy’s status. In figure 3.2 agents A and C are asked to confirm their requests to attack/continue the battle on agents B and D respectively. In the figure it is assumed that the agents C and D have been in battle at least one simulation time interval, and hence agent C’s request is implicit.

Depending on the attacking and defending agents’ status the attacking agent can decide either to confirm or cancel the request.

- The agent cancels the request, World treats the agent as no longer in battle and the agent will be notified about it through the Event parameter, described in section 3.2.1. The defending agent may or may not still be in battle, depending on if it is attacked by other agents.

- The agent decides to confirm the request, both the attacking and the defending agents are considered as in battle, and will be informed about this through the Event parameter. Agents which are considered to be in battle will be judged when all agents have simulated their next time interval.

- There is also a case when the World can deny a request, when a requesting agent already is attacked itself. In this case World treats the request as if the attacking agent had canceled the request.
3.2.3 Enemy Detection

Enemy detection is an important task every agent has to perform in some clever way. This task is about discovering enemies being nearby the troops themselves. For that purpose each agent has an associated attribute offering the World information about how far a particular agent can “see”, i.e. their surveillance distance.

Due to the choice of implementing nationalities, playing the role of a military base for each side, these nationalities are provided with another, somewhat different type of enemy detection — that from the air. Naturally, a greater area can be searched this way.

Since the World module has a complete knowledge about all the agents in the simulated world, it is also our choice of where to put the function of enemy detection. For increased efficiency this choice has turned out to be the best, instead of the perhaps more natural one of letting all agents (as separate processes) and nationalities do their own enemy detection procedures. This would require either each agent to keep a local database over all other agents in order to detect them, or to increase the communication between agents in order to collect all necessary information needed for a complete enemy detection. Clearly neither of these solutions are particularly efficient.

It is important to mention that enemy detection provided by World is not the complete enemy detection, but an information source, where agents and nationalities are given their possible enemies. It is thereafter their responsibility to calculate (with the use of tailored algorithms) the set of discovered enemies. This leaves the World module with the relatively simple task of checking the area within a certain distance and select all known enemies within that distance.

Neighbourhood detection

The work of enemy detection at the agent level is performed just after the attack confirm protocol, as can be seen in figure 3.1. It is an entirely separate function, apart from everything else in the World module. The following work is done:

1. For each agent $A$ get all other agents within the surveillance distance of $A$, which are also enemies of $A$. Collect them in an agent set $S$. By definition an enemy of $A$ is an agent which does not belong to the same nationality as $A$.

2. Associate the resulting agent set $S$ to agent $A$.

Computations performed by the algorithm regarding map coordinates and distances are done using primitives of other modules responsible for the terrain and map handling in the simulation system. The surveillance distance of a specific agent is obtained by asking the agent itself about the information.
Air detection

The work of enemy detection from the air is still another free-standing function, which propagates its result directly to the involved nationalities. The functions consists of the following steps:

1. Find all nationalities $N$ occurring in the simulated world. To each $N$ connect a bag, where the enemies will be collected.

2. For all agents $A$ in the world store the status of $A$ in the bag of all $N$, such that $A$ does not belong to $N$. This way all agents will be collected in all nationality bags where this agent is regarded as enemy.

3. Inform each $N$ about detected enemies by sending them the collected enemies in the bag.

3.2.4 World as a View Provider

One of the responsibilities of World is to make information available to the Clients. This responsibility is shared with the Nationality processes, and is therefore done via the common interface View (see section A).

Since World is somewhat a manager of the simulation (taking care of time, synchronisation, battle judgement, etc) the job as a View Provider seems both natural and easy. It is. Also, like one intuitively might expect, since the information stored in World is correct and up to date the information retrieved from World are facts.

In order to explain how World manages its role as View Provider, we use the same examples as introduced in section A.

Example 3.2.5 If a Client requests all known agents from the World View, as shown in example A.0.1, we simply extract all agents (with corresponding status) from World’s database, and return those. As indicated above, it is a straightforward task.

The notion of subscriptions, also introduced in section A, makes life for World a bit more interesting. In some way we need to store Clients which have subscribed to each event. This is done with an internal database in World. Thus, the responsibility of subscriptions is handled as in the following example.

Example 3.2.6 Assume that a Client $C$ subscribes for notification on event agent\_dead as in example A.0.2. Then it is up to the World process to deliver information concerning any agent that dies. Solution as follows: Store $C$ in the subscription database associated with agent\_dead. Every time World is informed (by Master) that an agent has died, World checks the subscription database for the event agent\_dead, and for each subscriber of that event sends him the notification. Thus, $C$ will get an event notification.

---

2As opposed to the information from a Nationality process, which are influenced by the acquired knowledge of that nationality. It might be neither up to date nor complete.
Chapter 4

Terrain

by Anna Erikson, Anders Johansson, Pär Mattsson, Martin Wikborg

4.1 Introduction

This paper describes the implementation of the terrain which is the representation of the world during the simulation in which the agents navigate, move and act. The terrain is two-dimensional.

The terrain is based on a real map in the t5 format, see 4.2.1. By using a GIS (Geographic Information System) tool named GRASS (Geographic Resources Analysis Support System) the map is converted into a digital map representation in ERLANG.

In the simulation system there are many versions of the digital terrain. One of them is the real world in which the agents move, from now on referred to as terrain. Each nationality has its own version of the world, used for navigation, the map. The maps and the terrain have the same representation but are not always identical because different nationalities can have different knowledge about the terrain.

During simulation both the maps and the terrain can be modified separately. For example: A nationality release gas into an area. The gas cloud will be entered in the terrain and in the nationality’s map, but not in other nationalities’ maps because they do not know it has happened until they have visited that area.

Since the module must take care of tasks both before and during the simulation this module description conceptually consists of two parts:

- Creating a terrain: Converting a real map in t5 format into a digital map in the internal ERLANG format.

- Handling the terrain during the simulation: Perform changes, respond to questions and maintain the robustness of the terrain.
4.2 Terrain representation

The internal representation needs to satisfy several different conditions to be useful.

- Map references (questions about the capabilities in the terrain at a certain point) must be done efficiently to avoid bottle necks.
- The terrain must be small enough to fit the primary memory during the simulation.
- It must be possible and efficient to navigate.
- It must be possible and efficient to make changes in the terrain.

We found that this did not hold for GRASS's own representation. Therefore we converted the T5 file to an ERLANG representation.

4.2.1 T5

To convert the different files (see figure 4.1 and 4.2) in T5 format, we use GRASS, as mentioned before. It is started by using a script with a couple of arguments. The arguments are in the file Initfile (4.4.1) and specifies which files to read and some other information.

The files to be read are in one of the following formats:

- Raster files (see figure 4.1)

  These files are dependent on the resolution specified in the Initfile, see 4.4.1. GRASS automatically truncates the raster bits to fit the resolution.

- Vector files (see figure 4.2)

  They are not affected by the resolution, but maintain the same size even if the resolution variable has changed.

These files are to be converted like raster-to-ASCII and vector-to-ASCII. To do that some of GRASS's convenient commands are used.

4.2.2 ERLANG representation

The specification suggested by CelsiusTech IT AB [8] had a hexagon representation with sides indicating roads and waters. Conceptually, roads lead from the center of the square to the middle of the side while waters run along the side. This is a model well-known in strategic games.

Since most real maps are built by squares we decided to adapt the hexagon structure to squares. Each of its sides has information about the roads and water connected to the square. Roads lead from the center to the middle of the side while water run along the side. Map references are made by indexing a row and a column.

A terrain square is represented by a 16 bit number in ERLANG where:
north: 6525000  The number of meters
south: 6500000  from the equator.
east: 1575000  The number of meters from
west: 1550000  the Greenwich meridian.
rows: 13
cols: 13
1 3 4 1 3 3 4 1 4 3 3 3 4
4 4 3 3 4 4 4 3 4 4 3 3 4
1 4 4 5 4 4 4 3 3 3 3 3 4
3 3 4 3 4 3 4 4 4 4 4 4
3 4 4 3 6 3 4 4 3 5 4 1
4 4 4 4 5 4 3 4 4 5 4 4
3 3 4 3 4 4 3 4 5 3 4
3 4 3 3 4 3 4 3 3 3 3 3
3 3 3 3 3 3 4 4 4 5 1
6 3 3 3 3 3 3 4 3 5 4
3 3 3 3 3 3 4 3 3 1 4
6 3 3 3 4 6 3 3 3 4 1 1
1 3 3 4 4 3 3 3 3 1 1

Figure 4.1: Example raster outfile.

- bit 0-6 stores the terrain type
- bit 7 denotes forbidden/not forbidden
- bit 8-11 denotes road sides
- bit 12-15 denotes water sides

The simulation uses five different terrain types: open terrain, sparse forest, dense forest, urban area and water. Since there are six bits reserved for terrain types it is possible to have up to 128 different types.

A single bit, denoting forbidden terrain, is set (to 1) when the area is forbidden and unset (to 0) when the area is restored again. The reason for this representation is that the rest of the information in a terrain type square will not be lost when setting or unsetting a square to forbidden.

The water and road bits denote which sides of the square contains water and which sides are crossed by roads.

A map (which basically is a grid of squares) is internally represented as an array with two dimensions. In ERLANG this is naturally implemented as a tuple of tuples.

This representation of the terrain fulfilled the criterias mentioned above. Indexing and entering changes in ERLANG tuples is fast and simple. Often this solution gives low efficiency because of data copying but this is not a big problem in ERLANG. The terrain is small and fits in the memory. It also is efficient to navigate in the terrain, see Chapter Navigator 5.
ORGANIZATION: LMV
DIGIT DATE: 2022-02-06
DIGIT NAME: razor
MAP NAME: roadmap
MAP DATE: 2012-02-05
MAP SCALE: 100000
OTHER INFO: Created by cyberdyne.arc
ZONE: 0
WEST EDGE: 1550000
EAST EDGE: 1575000
SOUTH EDGE: 6499999
NORTH EDGE: 6525000
MAP THRESH: 0

VERTI:
L 5

This section consist of 5 coordinates.
6525000 1565223
6524979 1565210.25
6524972 1565202.75
6524962 1565188.75
6524944 1565147

L 3

This section consist of 3 coordinates.
6525000 1565124.25
.
.
.

Figure 4.2: This is a part of the road file used by our version of the system.
4.3 Integration in the simulator

All troop movements are made in the terrain while troop navigation is performed in a navigation map, see Chapter 2 and 5. The terrain and the map has the same representation, but not necessarily the same information.

The interface to all terrain services are collected and available through a module named map. Some of the services are also available through the module terrain which just passes on the information to the map module.

All services are encapsulated in function calls to make them independent of the internal representation. This is the standard procedure to handle messages in ERLANG.

4.3.1 Logging

The terrain is maintained by a registered process which can be located at any node when using distributed ERLANG. It has an internal state which must be restoreable during simulation. The state consists of the original map and all changes made since start. All changes are immediately written to the log file.

A log file is ordered by the system supervisor. The supervisor makes the registered process write all its changes to a file. At any time it is possible to resume at a specific point by giving the original map and a file of changes. It is also possible at any time to save the whole current terrain.

The changes are also stored with another format to enable a user to get up to date with the current terrain. This information is not complete but enough for the users graphic display.

4.3.2 Queries

The most frequent service, asked by the agents, is answering questions about the terrain at a specified coordinate. This service had high priority when designing the system, since it must be done quickly. It is done by easily recalculating the coordinate into a map index and then make a map reference.

To minimise costly function calls, there is also a compound query to check out a whole set of map indexes. It is used by the navigator to update a whole area, see Chapter 5.

4.3.3 Changes in the terrain

It is necessary to be able to make changes in the terrain. All changes are stored permanently in the log file. The following services are available at any time:

- Change the terrain type in a square or a whole area, with or without removing all roads and waters.
- Set or unset an area to be “forbidden”.
- Insert a road or water between two points.
4.4 Implementation outline

Before the simulation we convert the original map, in T5 format [3], into our internal representation. The conversion is done by first using GRASS and a file Initfile for converting the raster and vector files into ascii format. Then the terrain types in the ascii files are changed as to fit the terrain types that we use presently. At last, we add water and roads to the map with the correct terrain types, thus making it complete.

4.4.1 GRASS

The first part of the conversion is made with help of GRASS which read the original standard format T5.

To use GRASS files, one have to have a GRASS mapset. The mapset is a workspace where the user can put the users converted files. Our system is not dependent on whether the user has run GRASS before or not, that is, if the user has a mapset or not before. If the user has a mapset, just start the startup process. If that is not the case a mapset is created for the user and a file .grassrc is created in the users root home-directory, $HOME. That file contains a couple of environment variables that GRASS reads from when it executes its commands. The file is specified more in detail in section 4.4.1.

The startup process is started when the user has got a mapset. Then the startup script is used with the arguments specified in the Initfile (see section 4.4.1 for a deeper explanation about the syntax). There the user specifies which files and which resolution, if they are raster files, are to be used. If the Initfile is correct the script checks if all files exist in the GRASS’s or the users mapset\(^1\), then start the converting of the files. If the superuser wants to change the game environment, like having a more detailed map or change the map, just edit the Initfile.

When we have read the initfile, we use GRASS’s commands to convert the maps. There are two kinds of conversion: raster-to-ASCII with different resolutions and vector-to-ASCII. The vector-files are not affected by the resolution.

After the conversion has been made, GRASS is shut down. After that first part of conversion four files lay in the directory from where the user started the script. It is one raster-file, two vector-files and a file for the map to be displayed.

.grassrc

This file specify where GRASS and it’s database are located in the system. So if you want to run the startup script on another system please locate GRASS in /usr/sup. Figure 4.3 show the .grassrc file.

\(^1\)If the map is in GRASS’s mapset, be sure to locate the specified file in GRASS’s directory data/9HSV/PERMANENT. Otherwise the script won’t find the files.
GISDBASE: /usr/sup/grass/data
LOCATION_NAME: 9HSV
MAPSET: d91ajo

Figure 4.3: .grassrc

The Initfile

The user must specify an initfile which tells the script what files are to be read and with what resolutions. The syntax of the initfile is specified in Figure 4.4. There is also an example in Figure 4.5.

RESOLUTION_TERRAIN: \( \Rightarrow \) <space>* INT <space>* COM
RASTER: \( \Rightarrow \) <space>* FNAME <space>* COM
RESOLUTION_MAP: \( \Rightarrow \) <space>* INT <space>* COM
VECTORROAD: \( \Rightarrow \) <space>* FNAME <space>* COM
VECTOR_MAP: \( \Rightarrow \) <space>* FNAME <space>* COM
VECTOR_WATER: \( \Rightarrow \) <space>* FNAME <space>* COM
INT: \( \Rightarrow \) <Digit> INT
FNAME: \( \Rightarrow \) STR
COM: \( \Rightarrow \) STR
STR: \( \Rightarrow \) <Letter> STR
Digit: \( \Rightarrow \) 0..9
Letter: \( \Rightarrow \) a..z | A..Z

Figure 4.4: The grammar for the initfile.

RASTER: topografi
RESOLUTION_TERRAIN: 50
RESOLUTION_MAP: 75
VECTORROAD: vag
VECTOR_MAP: vag
VECTOR_WATER: vatten

Figure 4.5: Example of the initfile.

4.4.2 The Implementation of conversion

In the simulation model suggested by CelsiusTech IT AB [8] the following terrain types were specified:
- Open terrain
- Sparse forest
- Dense forest
- Urban area
- Water
- Forbidden

In the original map there are many more terrain types than in the one used in the simulation model. Therefore all terrain types are converted into one of the above according to an initialization file created by the user, see Figure 4.6. The terrain type "forbidden" indicates an area where no one is allowed to be. "Forbidden" is also the default terrain type if an undefined terrain type is found during the converting process. The conversion is simply a translation from one terrain type to another.

The conversion program needs two files, one with the map and one which specifies how the map shall be converted.

| 0:4   | \ \ Comment         |
| 1:3   |                     |
| 2:6,2 | \ \ Comment         |
| 3:1   | 4:5                 |

---

Figure 4.6: The convert file our system is running.

**The conversion file**

The conversion file consists of rows with numbers, see figure 4.6. The number before the colon is the type the user want to convert to and the number(s) after the colon is (are) the type(s) converted from. The user can specify the conversion in different ways, as seen in figure 4.6. A comment can also be placed at the end of the row if wanted.

The converting program reads the convert file and parsed it to get all the terrain types and their mappings. The parsing is done by first using **FLEX** for the lexical analysis and then parsing the lexems in a c-program. All terrain types are stored in an array with their input values as corresponding index in the array. Their output values consist of their mapping values. See figure 4.7 for an example.

**The map file**

After parsing the conversion file the program parses the map file, given as output from **GRASS** (see Figure 4.1), and convert it. This is done by
terrain_type[1] = 3
terrain_type[2] = 2;
terrain_type[3] = 1;
terrain_type[4] = 0;
terrain_type[5] = 4;
terrain_type[6] = 2;

Figure 4.7: Example of the terrain array our system is using.

a C-program for the first six rows of the file (since these will always look the same), then both by FLEX and C for the rest of the numbers (the actual map). The numbers representing the different terrain types are read number by number, looked up in the conversion array and, if the number exists there, the corresponding value is printed. Otherwise an error message is printed on the LOG file, see section 4.4.2.

The converted values are printed in an ERLANG format, that is, in a tuple. (Except for the first value which is the size of a map square.) See figure 4.8 for example of the tuple representation.

Figure 4.8: A raster file, converted and represented in ERLANG tuples.

The LOG file

The result of the conversion is written into a log file named LOG. The following answers can occur:

- OK
If the conversion was executed without any problem.

- **NON_EXISTENT_TERR_TYPE**

  If the kind of terrain the user tries to map to doesn’t exist in the specification.

- **NON_EXISTENT_IN_CONV_FILE**

  If the map file contains some kind of terrain type that the user hasn’t specified in the convert file.

After converting the original terrain types, the result is used as a base for the internal terrain structure. On that base the roads and water are added to the terrain in a way that will be described below.

Figure 4.9: An example of a vector object of the original type
4.4.3 Road layout

The vector file representing the roads must be processed and added to the data structure. The original roads are represented as a sequence of coordinates, where the roads is straight between the coordinates (see figure 4.9) and will be converted into a sequence of square indexes.

![Figure 4.10: The example vector interpreted as a road](image)

The principle of the algorithm is to find every crossing point between roads and square sides. A crossing point is found if a road section goes between two consecutive squares. Two squares are consecutive if they have a common side.

The crossed square side will then be marked with the attribute of having a road from the middle of the side to its center. This will be done for both of them.

By iterating through the sequence of coordinates a match is done for each road section. Basically, a crossing point is found if a road section goes between two consecutive squares. The iteration is continued until all road
sections have been processed.
Each road section is tested whether it crosses a square side or not. There are three possible cases:

- The road section is totally included in a square: It will have no influence on the layout.

- The road section goes between two consecutive squares: A road side has been found. The squares will have their common side marked as a road.

- The road section goes between two squares which are not consecutive: We divide it into two parts and try again. At the end there might be a situation where the road section has almost no length, but still lies in two unconnected squares. This indicates that the road passes through a square corner. Since roads only can pass through square sides a choice must be made to decide on which sides to put the road. The choice is generally arbitrary but must include two sides, one north-south and one east-west to make the road continuous.

4.4.4 Water layout

The representation of the water is similar to that of the roads. The water vectors from GRASS follows the contours of every type of water in the terrain. For that reason, even slim objects like creeks are represented by a water contour.

To improve the final result all open areas of water are equipped with a complete set of water sides.

As in the road layout algorithm the search is focused on finding crossing points between vector objects and square sides. The main difference between roads and waters is that water runs along the side of the square while a road crosses the side.

Since waters run along the side of the square the interesting thing about the crossing points is not the side which has been crossed but the closest corner of it. When a crossing point is found the closest corner of the crossing point is calculated.

As with the road layout algorithm there are three cases possible when examining a road section.

- The water section is totally included in a square: It will have no influence on the layout.

- The water section goes between two consecutive squares: A possible water side has been found. If the closest corner of the crossing point is not the same as the previous corner found a water side layout will be performed. If the two corners compared are identical the crossing point will have no influence.
Figure 4.11: The example vector interpreted as water
• The water section goes between two squares which are not consecutive: The section is divided into two parts which will be next on line for testing. If the water section passes through a square corner that corner will be considered to be a crossing point.

When two different corners have been found a water side layout is performed. The two corners always belongs to the same square. There are two possible cases:

• The corners are connected by a square side: That side will be made a water.

• The corners are placed diagonally across the square: Two sides (one north-south and one east-west) must be set to connect the two corners. The choice of sides is basically arbitrary but in this implementation an improvement is made to increase the quality of the water layout.

The trick is that one should try to set as few sides as possible. Previously set sides thereby are re-used. This is the reason why all areas with water initially are given a full set of water sides.

All water areas also has a water vector connected to them. When one tries to minimize the number of water sides the water contours will be better placed according to open water areas and previously made water insertions.

4.4.5 Lookup and changes

During the simulation the terrain is taken care of it’s own process. This process will receive messages from other process which want to have information about the terrain. The terrain process gets the square corresponding to the coordinates sent by the asking process and sends back an answer which is the terrain in the square.

A square’s index is calculated by dividing the square size with the coordinate. Example: If we have the square size 100 and the coordinate is 12444,987 the result will be 124,9.

The terrain process also receives messages that will change the terrain. The process calculates the right square and enter the terrain change in the indexed square. Changes are stored for the logging facilities.
Chapter 5

Navigator

by Daniel Blomqvist, Magnus Ingelbo, Olle Pellijeff and Peter Öhman

5.1 Introduction

In order to complete their tasks, the troop units (Agent Module (see Chapter 2)) need to move around in the terrain. The problem is to find a route which corresponds to the troop movement request, which is one of the following routing tasks:

- Find a route from the current position to a goal position.
- Find a route which takes the troop a given distance away from a given position.
- Find a route from the current position to a given position while avoiding a position with a certain radius.
- Find the fastest route from the current position to a given position.

The Navigator Module provides the troop units with routes, satisfying constraints regarding protection and time. Each nationality has a private version of the map with an associated navigator that will perform navigation for troops of that nationality. As a troop discovers changes in the true world, the map for its nationality is updated. This corresponds to the real-life use of a map to plan movements.

The actual route planning is done with a process-oriented method, where a group of processes strives to construct one of the routes specified above. Each process, called a tracker, performs a local search\(^1\), based on adjacent positions on the map. A weighted function of the constraints from the navigation request gives a measure of success for an individual tracker. Only the tracker with the best success value is allowed to move. Visited positions and the measure of success is communicated to other trackers, which can

\(^1\)The term “local search” also occurs in the field of Constraint Programming. We use the term in more general meaning.
avoid already visited positions. When the goal is found, a post processor is
examining the route, and finally, the route is delivered to the agent. The
search is supervised by a tracker handler, for example, handling spawning of
new trackers. The tracker handler also limits the number of active trackers.
The search is guarded by a watchdog, which will abort the search after
a specified time limit. The trackers thus behave like interactive agents\(^2\),
working together towards a common goal.

During the development it became necessary to visualise the behaviour
of the trackers. For this purpose an extensive test-bench was constructed
which gives a graphical overview of the trackers and the terrain. It also
allows tuning (tweaking) of the parameters involved in the tracker algorithm.

## 5.2 Problem Specification

### 5.2.1 Map Format

The map format is a grid, where each square in the grid has homogeneous
terrain. Each terrain type has passability and protection properties. Each
of the four edges has associated information about roads and rivers. Rivers
follow the edges, roads lie between edges. The Terrain Module (see Chapter
4) provides a map in that format.

### 5.2.2 Navigation Services

The routes returned is made up of subsections, specified by a pair of coor-
dinates. Each section has homogeneous terrain and a road-flag, indicating
if a road exists in that section. Troop units can request different types of
routes.

- **Path:** This is the generic case where the troop unit requests a path
  from a current position to a destination position. A time limit, the
  maximum time it should take to travel the route, is also given. The
  route returned should maximise protection given the time constraint.

- **Around:** Around is a special case of the path request where the troop
  unit requests to move to a destination position, but also needs to avoid
  an area, specified as a position and a radius. The route returned should
  maximise protection under the time constraint.

- **Towards:** Here the troop unit requests to move as quickly as possible
  from its current position to a destination position. A time constraint
  is also given. The route returned should minimise the time it takes to
  travel the route.

- **Away:** Similar to the above, but the troop unit instead requests to
  move from its current position away from a given position. A destina-
  tion position is not given, instead the troop unit specifies how far it

\(^2\)The agents here should not be confused with the troop units that are called agents in
the Agent Module
wants to move away. The route returned should minimise the time it takes to travel the route.

5.2.3 Reconnaissance

Troop units wants to do reconnaissance. The unit can request a search within a radius of their current position. All differences between the map used by the units nationality and the true world found in this area is added to the nationality’s map.

5.3 Solution Outline

The route planning is done with a process-oriented method, where a group of processes strives to construct one of the routes specified above. Each process, called a tracker, performs a local search, looking only at adjacent positions. One algorithm handles all types of navigation services. Different input parameters to the tracker algorithm controls the choice of navigation service. The tracker moves between center of edges, crossing one square (see Figure 5.1). The different trackers competes, so that the first one that achieves the goal, returns the route.

The trackers are coordinated by a tracker handler, so that there is an upper limit on the number of concurrent trackers. The tracker handler also provides multicast functionality to the trackers. A watchdog limits the total time the search may take.

![Figure 5.1: Moving from edge to edge](image)

5.3.1 Interactive Agent Approach

The group of trackers behaves like interactive agents, fulfilling the three criteria below.

- **Common Goal:** During the handling of a navigation request, each tracker is started with the same goal.

- **Intercommunication:** Whenever a tracker visits a position, that position is communicated to the other trackers. So, every tracker has a
set of visited positions that it should not use. Since the trackers are competing to the goal, a measure of their success is also communicated.

- **Identical Behaviour**: Every tracker uses the same algorithm.

## 5.4 Implementation

We chose to implement the algorithm in ERLANG, as opposed to C, because we wanted to investigate if ERLANG is a suitable language for this kind of problem.

The map is internally stored as a tuple where each element is a tuple containing the integer representation of the squares in one row. Access to squares is fast, updates are more expensive but rare.

### 5.4.1 Control Flow

Each nationality in the simulation has a navigator, implemented as a process that is running under the entire simulation. Navigation requests are made to the terrain process, which sends them on to the navigator belonging to the nationality of the troop unit issuing the request. When the navigator receives a request it spawns a new process, a tracker-handler that administers the search. The tracker-handler can create trackers, provide multi-cast services to the trackers and kill all trackers when a result is reached. Figure 5.2 shows a simplified picture of the control flow.

![Control Flow Diagram]

**Figure 5.2**: Description of control flow

### Navigator

The navigator is the administrator of navigation services for one nationality. When started it reads a map from a file. This map, with possible updates, is then regarded as the map known by the nationality that use this navigator. The navigator keeps a log of changes made to the map, so that it is possible to restart the navigator at a given state. Upon a request for a search in the map the navigator spawns a tracker-handler process and thereafter enters a wait state, waiting for an answer from the tracker-handler.
Tracker Handler

The tracker-handler administrates the trackers. It initially spawns a watchdog and a tracker. When a tracker is entering a new square in the map the tracker-handler is notified. The tracker-handler then multicasts a message to all other trackers. Trackers can also request to spawn a new tracker, which will be done if the maximum number of trackers is not exceeded. The search terminates if a tracker reaches the goal position, the watchdog times out or all trackers died. When one of these situations arise, the tracker-handler is notified, whereupon the tracker-handler kills all trackers alive and the watchdog, sends the result to the navigator process and terminates.

5.4.2 Tracker Algorithm

The algorithm described here is actually the case for the path navigation request, see 5.2.2. This case is, however, the most generic and the other cases has only slight modifications. A tracker's state consists mainly of its current square, edge, set of visited positions and success value. At each position, it considers the squares straight ahead, left and right, moving across the current square (see figure 5.3). Exits blocked by unpassable terrain or already visited are then removed. All exits could be removed, thus forcing the tracker to backtrack. The remaining exits are evaluated and the exit with the highest value is chosen. The other evaluated exits are then compared with the chosen exit to determine if it is worth spawning a new tracker to each of those exits. A set of parameters (adjustable in the test-bench) controls the behaviour of the tracker.

Angle Calculation

The angle for an exit is calculated as the the angle between the direction to the target and the direction that the tracker would be facing after moving to that exit (see Figure 5.4).

![Figure 5.3: Moves considered at each step](image)

Exit Evaluation

Each exit is evaluated and assigned a numeric value. The value is a weighted sum of protection, speed and angle to target, all taken from the square which that exit leads to. Exits with angles larger than 90 degrees are discarded, so that the tracker will not move away from the goal. The weights can be set when using the test-bench.
Follow Water

When water is blocking a tracker’s movement towards the goal, it needs to follow the water-edge, moving away from the goal (which is not normally allowed). A tracker will continue to follow water as long it cannot move towards the target. This is implemented as filtering out exits which does not lead to positions adjacent to unpassable terrain.

Spawn Control

The tracker always chooses the exit with the highest value from the exit evaluation. Each of the remaining exits are then compared with the chosen exit, so that a spawn takes place if their values differs less than the spawn parameter.

If two compared exits lead to positions adjacent to unpassable squares, a spawn always takes place to get the follow-water behaviour. To prevent unnecessary spawning when moving along a border between two terrain types, a spawn does not take place if the terrain-types has not changed. A change of terrain types for two exits means that they are different from the terrain types at the corresponding exits at the trackers previous position.
If no possible exits exist, even away from the goal, the tracker undo its latest move, keeping the current position as visited. This will force the tracker to try a different exit at the previous position or backtrack even further. If it needs to backtrack at the starting position and no possible exits exist, not even away from the goal, the tracker undoes its latest move and keeps the current position as visited. The best move keeps the current position as visited. The best move keeps the current position as visited. The best move keeps the current position as visited. The best move keeps the current position as visited.

Post-processing

Post-processing is done due to the fact that the trackers can make some unnecessary turns which are removed. The shortcut is made by searching through the path. If it is found that a position later in the path is adjacent to the present position and there is not a water-edge between these positions, then the unnecessary part of the path is dropped, see Figure 5.6. The path is finally converted from our internal representation to the format specified by the Agent module, see section 5.2.2.

Test-bench

The test-bench was developed in Tcl/Tk with an interface to Erlang. The test-bench is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17]. The test-bench is shown in Figure 5.6. It is written in TCL-Tk with an interface to Erlang [17].
- **Watchdog time**: Total allowed (real) time for a navigation request.
- **Tracker nr limit**: Maximum number of trackers allow during a navigation request.
- **Distance**: For the away and around navigation request, this parameter specifies the wanted radius.

There is a different set of weights for each type of navigation service, and the context can be switched in the supervisor window described in the next section.

![Preference window for the test-bench](image)

**Figure 5.7**: Preference window for the test-bench

### 5.5.2 Supervisor Window

The supervisor window is divided in two parts, a command section and a section displaying the search.

The command section is shown in figure 5.8. The buttons titled **Cmd**: is used to choose type of command, and the buttons titled **Pref**: is used to choose weights context. The label **Pos**: shows the cursor's current position in the search displaying area, the format is \((\text{row}, \text{column})\). The label **Path**: shows the current navigation service. At the bottom of the area the colour codes for the different types of terrain is showed.

The search display area, part of which is shown in figure 5.5, is showing the map and the trackers performing the search. Figure 5.9 shows a description of symbols used to display the search.
5.6 System Integration

The interface to all services is collected in one module, Terrain. Nationalities is registered when created, and a navigator is started for each nationality. The terrain process then passes on requests to the right navigator.

5.6.1 Distribution

A navigator can be located at any node in Distributed ERLANG [17]. The processes spawned by the navigator under a navigation request is local to the ERLANG node.

5.6.2 Logging

Both the terrain process and the navigator processes have internal states that need to be logged, if a simulation is to be resumed.

The terrain’s state is the mapping from nationalities to navigators. This state is saved as a list with tuples containing a nationality/navigator-pair.

The navigator’s state is the filename of the navigator’s initial map and changes made to the map. When restarting the map is read from the initial file and all changes saved in the log are made to the map.

5.7 Conclusions

The amount of code in ERLANG is small relative to most other languages, this has been a great advantage to us during the project. Thanks to ERLANG
it was very easy to build a working prototype, so that we at an early stage could test our ideas. The process model supported by ERLANG made it completely trivial to implement concurrency and modularisation of the simulator.

5.7.1 Quality of service

Analysis of concurrent algorithms is in general hard, and the ERLANG environment do not currently have any good profiling tools.

In most cases when there is a way to reach the goal, a path is found. The paths found is not optimal in the sense of protection. During the search a single tracker only considers its adjacent squares, which means that it can miss possibilities to seek protection that lies further away.

There are cases when our search fails to find a possible route. This is due to problems in handling local maximum, when the map contains a concave border (in our case against water or forbidden area) and the goal direction is unfavourable, see figure 5.10. In this case the tracker leaves the border and tries to go in the goal direction. This will kill the tracker because of the constraint that a tracker must not walk away from the target, except when following a border. If no other tracker in our search is able to find an alternative path, the search fails, even though there may have been a way around the water.

![Figure 5.10: A situation where this tracker will fail](image)

5.7.2 Execution speed

During the simulation more than one hundred processes are active. Under this load, and with the computers we use, JAM ERLANG does not provide the necessary speed. When asked for a long and/or difficult path, the watchdog time limit, currently set to two minutes, may be exceeded.

It is possible that another implementation of ERLANG could give a speed up.
5.7.3 Suggestions to improvements

The tracker could look further when choosing an exit. This would enhance its ability to seek protection.

Trackers could be started from both the current position and the goal position. The trackers could attract trackers going in the other direction, so that when they meet, their individual paths combines to the final path.

5.7.4 Alternative approaches

An alternative approach, using global information, is to use a divide-and-conquer algorithm, trying to make straight routes between to start and goal position. When a route is blocked by a object, the algorithm would try to make subroutes by searching orthogonally from the center of the object, in both directions. The algorithm could be used in several passes, with different levels of protection for the blocking objects, thus maximising protection for the route returned. A problem with this approach would be to handle all navigation requests, for example away which has no goal position.
Chapter 6
Master

by Arne Borälv, Gustaf Holmkvist, Martin Klint and Herman Ägren

6.1 Introduction

The sim94 project is intended for training of order-giving of military personnel, by concurrent simulation of troops in a dynamic terrain. The simulation consists of autonomous, communicating processes, implemented in the high-level language Erlang. These processes, interacting with each other, makes up the whole simulation system.

Since there is no way for a process to start itself, or no global data, a process that manages and supervises the system is needed. This is the Master process.

The Master initialises the system. This involves creating a framework for a distribution model, and starting up processes that will hold data needed for managing the system.

When the system has been initiated, it can be viewed as a server which players can connect and disconnect to as they like. All connections are made by the Master module via an UNIX socket [2].

During a running simulation, the Master performs requests from processes in the system, and from clients logged on. These requests are creation/deletion of processes, saving/loading of a simulation state and information about what the system looks like.

In order to make error recovery easy, and to gain more efficiency, the processes are distributed. Error recovery can be handled transparently, by keeping a runtime-log and, in case of partial failure of the system, restarting the system from the latest logged state.

To configure the system in a way so that maximum advantage is taken of the CPUs involved, the Master provides migration services, together with statistics on a per-machine basis.

The Master also shuts down the simulation in a controlled manner upon request to do so.
6.2 Preliminaries

ERLANG is a concurrent, functional language originally developed for AXE telephone-switches, by Ellentel [7]. The Master module is entirely written in ERLANG. For a detailed description of ERLANG, (see Section 1.3).

A process is a program state. Processes execute concurrently in ERLANG, where communication and synchronisation are done by message-passing and matching. The ERLANG system guarantees that messages sent are delivered.

The simulation is distributed, which means that the computations, e.g., the processes, are executed concurrently on several machines. A distributed application have a potential to run faster and more reliably than the corresponding non-distributed application. However, the number of error sources drastically increases compared to non-distributed applications. Nodes may fail, network connections can fail or their capacity might temporarily be degraded. With powerful tools such as distributed ERLANG [17], it is possible to write distributed applications almost as easy as writing non-distributed ones.

Statistics, or load, is (for now) measured on a per-machine basis. The measurement considers how the past load has been (how many reductions the node has made), and what the predicted future load is (the length of the processor run queue).

Migration of a process means that the process is moved from one machine to another. With statistics and migration services, it is possible, during a running simulation, to configure the distribution in a way such that maximum advantage of the distribution is taken.

Distributed applications are often written using a library of network access functions, which access the network in a controlled manner. The Berkeley 4.2 BSD UNIX socket [2] interface is one example of such network functions. We used ERLANG’s built-in socket interface for UNIX sockets.

Logging on to the simulation means setting up a connection to the simulation servers’ socket, over TCP/IP [13]. TCP provides a reliable flow-controlled, two-way transmission of data. TCP constructs virtual circuits between peer entities. A virtual circuit consists of remote Internet addresses, remote ports, local Internet addresses and local ports. IP uses the Internet addresses to direct messages between hosts, and the port numbers to identify a TCP entity at a particular host.

Logging of the simulation state means saving the states of all processes (see Section 6.3.8). In case of a node failure, we do not start the simulation all over again, but roll-back (see Section 6.3.6) to a state which has been logged. Logging is done periodically.

A distributed system is robust when several nodes cooperate in such a way that failure of one or more nodes do not affect the functionality of the system as a whole.

All processes that are to be started by the Master module must be described by a Process Description, as specified in[4]. This Process Description says how the process is to be started, how it is to be logged and how to put
a new state to it. It also says if we are to register the process under a name. This process interface is the same for all processes the Master module starts.

Dependency-lists are associated with all processes in this system.

Example 6.2.1 Assume processes A, B and C will store the reference to process X, because they will all have communication with X. Then the dependencies are set by associating a list [A,B,C] with process X.

Updating a process state to a process P is done by applying a function in P's Process Description, with the new state as argument.

If this update of the process state is done after the process has been restarted, other processes are likely to hold an old address to this process, why these must be updated. This is done with dependency lists;

Example 6.2.2 Assume process X has the dependency-list [A,B,C]. Then, if process X is restarted (see Sections 6.3.8 and 6.3.6), we inform all processes in X's dependency-list, e.g. A, B and C, about X's new address.

The dependencies are set when the Master creates the process. The dependencies between processes are described as follows;

Example 6.2.3 Assume that all processes of type T1 will store the address to process X. Then, if a process P1 of type T1 is created, X is added to P1's dependency-list.

This way the dependencies are set once and for all, and restarting a process P always involve an automatic update of all holders of P's old address. If several processes are restarted, i.e. given new addresses, the dependencies are not sent until all new addresses have been calculated.

6.3 The Master Module

The simulation system is client-server based. Starting the Master process starts the simulation server. With this server running, the Supervisor logs on to the server via an UNIX socket, and starts an initial scenario with nationalities and troops.

All creation of processes in this system are made by the Master module upon request. There is a set of functions for creating processes, each corresponding to one type of process.

The use of an UNIX socket means that the players and the Supervisor do not need to be physically near the server.

When the Supervisor has made a scenario, the players can log on to the system, again via the same UNIX socket. This involves authentication by passwords. The Supervisor and the players can log out at any time during the simulation, and return later to continue the interaction.

Typically, the Supervisor wants to save a scenario, in order to point out an error made by a player, such as giving an erroneous order to a troop. The Supervisor informs the player about the error he made, and then restarts from that point. The Master module provides these save and load services.
In order to improve performance, the processes can be distributed on several machines (see Section 6.3.4).

The Master module provides commands for migrating processes, which is done through a drag-and-drop interface. To know which processes that are most computational-intensive, e.g. the ones that should be migrated (or even placed solely on a machine), the Master module provides statistics on a per-machine basis. The statistics measures past and future load of the nodes, and displays these in diagrams together with the UNIX load for the machine.

Since the simulation is distributed, there is a possibility of a machine or a network going down. These errors are handled transparently by logging the state of the system periodically. When an error occurs, the distribution is reconfigured, and the system rolled back to the latest logged state.

If the node where the Master process resides should go down, the system is rolled back by a stand-by process that monitors the Master from another machine.

The Master module also shuts the system down in a controlled manner, if instructed to do so by the Supervisor.

### 6.3.1 Implementation outline

Our aim has been to make an as general solution as possible. The functionality of the Master module is divided into two modules; the Kernel module, holding a collection of general functions, and the Master module, using the Kernel module to implement a simulation-specific interface. Thus we have achieved an abstraction, where the Kernel views the system only as a set of processes which it manages, and the Master offers a set of exported functions based on this abstract layer.

### 6.3.2 Startup of the simulation server

Starting the Master process starts the simulation server. The server consists of a number of processes needed by the Master process. These processes are numbered, starting from 1 to some number X. All processes created later are also numbered, from X and upwards. This ordering of processes is essential in case of roll-back, since processes must be restarted in the same order they were originally created (see Section 6.3.8). All numbers assigned to processes are unique in the simulations’ lifetime.

After the server has been initialised, the Supervisor can set up a scenario, and the game can begin.

### 6.3.3 Client communication

The client communication is kept as much as possible separate from the simulation. The reason for this is that we want to have a uniform way of manipulating processes when loading a saved simulation, migrating processes or handling node failures. One process controls the socket communication.
All socket communication from within the ERLANG system is done by the built-in socket interface [7] of ERLANG 1.

**Authority check**

Clients that log on to the system are checked for authority. Clients can be of two types, either a Supervisor, or a player, named after his nationality in the simulation.

One authority process monitors passwords for users trying to log in. The process uses the UNIX crypt [2] command to store the passwords (somewhat) secretly. The authority check consults a file containing encrypted pairs of name and password.

Initially, the game supervisor gives every nationality he creates a password.

### 6.3.4 Distribution of the system

The system can be distributed on several hosts, which gives higher performance and better error recovery. Distribution is easily implemented in Distributed ERLANG [17], that offers a large variety of high-level commands concerning this topic. The important issues are how to make the system robust and how to design the process distribution in order to get as high performance as possible. A robust system has a high fault tolerance, mainly because the different nodes in the system can control the others. If one node for some reason goes down, the other nodes can restart the lost processes (see Section 6.3.6). To create and maintain a design with high performance is a difficult problem. The Master process uses a naive heuristic for load-adjusting, that can be manually changed by the Supervisor.

**Implementation model**

The hosts that are to be used are given in a file, "hosts.erlang". When the simulation server is started, one node is created on each host. The system has a Master node, which always is the local node, i.e. the node on the host where the Master process was started from. The node names are generated automatically. Every node runs as a UNIX process.

The distribution is transparent. Processes can be registered and unregistered globally. The global registration facility uses relay processes on each node, that passes on the message to the real process. The relay processes make it possible for a client to send a request to a server without knowing at which node the server is. The client simply assumes that the server is on the same node, sends the message and then the relay process on the local node passes it on to the real server. For example (see Figure 6.1), if a client sends a request to a server named "foo", the message is first sent to the local relay process named foo and then sent to the real process. The relay process uses

---

1 Clients do not need to be ERLANG systems, however. In this system, the clients are written in STk [11, 16].
the Process Identifier of the real process, which is global [17]. There is also a relay process on the node where the real process resides. This might seem unnecessary, but it depends on the implementation. This solution makes it possible to treat all nodes in a uniform way.

To make sure that relay-processes really are created on each node after the global registration, the procedure is made atomic. This is done by each relay sending a confirm message to the process that called the global registration facility.

![Figure 6.1: Relay processes](image)

**Distribution design model**

By designing a distributed system here means to maintain a model of distribution that gives good performance. For example, processes that communicates a lot should be on the same node since communication over the net is slow. A process that does not have much communication but is resource-demanding, on the other hand, probably would do best if it were to run on a single node. Obviously, a big problem is to decide which processes should be on which nodes. Another problem is that the hosts on which the nodes reside usually have different load, and therefore different performance. Since the distributed system is highly dynamic (a node can go down, or get heavily loaded), it is a difficult problem to maintain a good design.

Master uses a simple model; when there are two nodes, the Master processes runs on the local node and everything else on the other. If there are more than two nodes, the Navigator processes (see Chapter 5) are started on the rest of the nodes (see Figure 6.2). The communication intensive parts, e.g. the World process (see Chapter 3) together with the Agent processes (see Chapter 2) are kept on the same node, and the computation intensive processes, e.g. the Navigator processes, gets an own node as far as it is possible (see Figure 6.2).

If a node goes down (see Section 6.3.6), the node with lowest load is always chosen as the node to restart the lost processes on.

In this distribution model, the Supervisor is given the control of how the distribution should be designed. By analysing the provided load statistics and explicitly migrating processes, a good performance can thereby be obtained.
6.3.5 Statistics

The Supervisor has a window displaying statistics for every node (see Figure 6.6). This is used for monitoring the system load, and to check if some node is very slow. If so, some processes need to be migrated from that node. A statistics-manager process on the Master node is used to retrieve the data needed for this window. It holds a history of each node over the last twenty measures. Every node has a statistics-collector process running, created initially by the Master. It has one period when nothing is done and one period when a dummy loop runs. The dummy loop is used to produce reductions, since a very fast node otherwise can do few reductions if all its processes are in receive mode. Right before the loop is started, the measuring is started and right after the loop is finished, the data is collected and the statistics manager is informed about the new data. Whenever new data is reported, it is displayed in the graphics window.

The relation between the period when nothing is done and the period when the loop is running can be adjusted.

Four different types of data is presented; Reductions/sec, Run queue, Unix load and Total load.

**Reductions/sec**

Whenever a function head is matched in ERLANG, a reduction is performed. Therefore, the number of reductions is a measure of system performance [7]. Many reductions per second means a fast node, probably with low load, whereas very few reductions can indicate an overloaded node. As mentioned above, reductions is not a good measure if a fast node is in receive mode, i.e. waiting for a message. Then no function calls will be made, which means no reductions. A dummy loop that always calls itself is used to overcome this problem. This of course reduces the performance, but the time it runs can be adjusted to be quite small in relation to the overall runtime, so it is not a big problem.
Run queue

The run queue indicates how many processes are currently in the schedule queue waiting to run. This is a good measure of the future load of the node [17]. You should probably not migrate more processes to a node with a large queue. The Run queue is always checked right before reporting the data.

UNIX load

To complement the other data, the UNIX load of a certain node (i.e. its host) is also showed. This is the load given by the UNIX command “w -u”. It is sometimes a more stable measurement.

Total load

This is the load that Master usually checks when doing decisions concerning the distribution design model, as previously described. The total load is the result of dividing reductions/sec with Run queue. This seems like a good measure since it combines the past load (reductions/sec) with the future (Run queue). In a way it shows how many reductions have been done for each process on the node.

6.3.6 Node failure

Assume that the processes on a machine dies due to a machine failure, and that some of them dies in the middle of a communication with processes on other nodes. Then, after restarting all processes that died, some processes might be dead-locked because they are waiting for a message from a process that no longer exists. Others still must be updated with the new addresses of the restarted processes. These are the problems that must be handled when a node fails.

Now, consider Figure 6.3.

The only way a runtime-error can be discovered in ERLANG is as a process exiting abnormally. This can happen of two reasons; either a bug in the code (during development of the system) or a network or machine failure. This is handled transparently, by rolling back the simulation to a state previously logged.

A node (or a network) going down is also discovered as a process exiting abnormally (see Figure 6.3). If we do have a saved log, we roll-back the simulation to that state. This is done by firstly killing the World process 2. Next, we find out which processes that are dead-locked, by a simple message-protocol. Those that do not respond within a specified time-interval are assumed to be dead-locked, why we kill them, then restart them together with the World process. Now, all processes are alive and in a stable state,

2Since it is likely that it has gone into a dead-lock because of it's heavy communication.
0: Simple request about simulation parameter
1: A process exited abnormally because of a bug in program
2: Hard request that require that we pause the simulation
3: Master node goes down
4: Node failure discovered

Figure 6.3: State diagram for master process.
why we can put the latest logged state to the simulation. Finally, all dependencies are updated (see Section 6.2.2).

There is also a possibility of node failure of the node where the Master process resides (see Section 6.3.6).

Algorithm 6.3.1 (Node failure) *Performed when a node has gone down.*

```plaintext
pause simulation
Dead = All processes on the crashed node and all deadlocked processes
NewIds = [ ]
FOR all process ∈ Dead DO
    Id = restart process on another node
    NewIds = append(NewIds, Id)
    send last logged state to process
OD
FOR i=1 to i=NumberOfProcesses DO
    send to process; all new Ids process; depends on
OD
resume simulation
```

Failure of a single process is considered as a runtime error due to faulty code. This will not happen in an error-free version of the system. But in developing and debugging the system, there is need for handling of these errors.

Algorithm 6.3.2 (Process failure) *A single process has failed.*

```plaintext
pause the simulation
Query the user
If answer = continue then
    resume simulation
fi
if answer = reload then
    reload the code of the failed process
    set the last logged state to the simulation
    resume simulation
fi
if answer = quit then
    quit the simulation
fi
```

**Master stand by**

In case of node failure of the node where the Master process resides, the Master process will die. This will result in all processes in the simulation

---

3The easiest way of implementing restart is to kill all processes, and then restart them, and finally updating their states to the latest log. But since certain processes has a significant startup-overhead, we chose to use this approach instead.
dying, since the Master process is linked \cite{7} to all processes in the system. However, there is always a stand-by process on another node. In case of the Master process dying abnormally, the stand-by process will restart the simulation from the latest logged state. For now, this will result in all clients logged on being thrown out of the system.

**Algorithm 6.3.3 (Master node failure)** If the Master node should go down.

Standby process discovers that Master process exited abnormally

Standby process re-initialises the simulation server

The last logged state is loaded

6.3.7 Pausing the system

As will be said below, some requests the Master services require that we have a paused system; e.g. the time has to be paused, as do client requests into the system. The time is paused by an asynchronous call to the World process. This call is confirmed when the World process has done all calculations for the current tick (see Chapter 3).

Client requests are paused by buffering all client requests in the socket process. When the request has been serviced, the buffered messages are sent, and the World process is resumed.

6.3.8 Request services

Consider Figure 6.3. The Master process does normally an idle wait for requests, and, upon receiving one, performs it. Some requests can be answered right away by a single message, or by a single lookup in a database. These requests are creating/killing of processes, and requests about simulation parameters.

Other requests require a paused simulation before performing them. These requests are process migration or saving/loading of the simulation state. Upon receiving such a request, it is added to a queue of requests the Master process holds, and the system is paused. Then we wait for the World process to acknowledge the pause. During this wait, other request may arrive. They are added to the queue, and, when the acknowledgement is received, we perform all the requests in the queue. Finally, we resume the simulation, and go back to the idle wait.

**Process creation**

When processes are created, the Master assigns consecutive numbers to them, that are unique for the processes in the simulations’ lifetime. This ordering of processes is essential in case of a roll-back of the system; creating some processes require that other processes already exist.

A process that is started in this system returns an abstract data type, an Id, which contains the process’ unique number.
Process migration

Migration is a request that requires that the system is paused. When the system has been paused, the process to be migrated is logged and then killed. Then it is restarted on the new machine, thereafter we put the logged state to it. Before resuming the simulation, we send the process’ new address (e.g. Id) to all dependents of it.

Logging

Before logging can be done, the system has to be paused. All processes are logged periodically, to get a robust system. This way, we need not start the simulation all over again in case of an error, but can roll-back to a state which has been logged. The system state log mechanism differs from the log used for migrating processes; processes must not log any addresses to clients logged on to the system when saving the system state. The reason for this is that clients can log on and off to the system as they like, why it is no point in saving their addresses in a system-state log.

Algorithm 6.3.4 (Log algorithm) Performed periodically.

```
pause simulation
FOR i=1 to i=NumberOfProcesses
    DO
        send request to process_i for its current state
        (except for client addresses)
    OD
State = []
FOR i=1 to i=NumberOfProcesses
    DO
        State_i = answer from process_i
        State = append(State, {i,State_i})
    OD
write State to file
resume simulation
```

Saving of simulation state

Saving of a simulation state is analogous with logging, except for that the saved state is kept in a file in a separate directory. The Supervisor baptises the file.

---

4 The reason for this is fairness in the simulation; some processes must not be allowed to continue their simulation while others are stopped.

5 Processes can hold an address to a client, because clients can subscribe to certain events (see Chapter A) in processes. For example, a player might want to be informed by his troops if they discover an enemy-troop or a blown-up bridge.
Loading a new state to the simulation

The Supervisor can load a previously saved state from a file at any time.

**Algorithm 6.3.5 (Load state)** Performed on request from the Supervisor.

- pause simulation
- kill all processes in the simulation
- read saved state from file given by the Supervisor
- Saved = All processes in the saved state
- NewIds = []
- FOR all process ∈ Saved
  - Id = restart process
  - NewIds = append(NewIds, Id)
  - send saved state to process
- FOR i=1 to i=NumberOfProcesses
  - send to process_i all new Ids process_i depends on
- resume simulation

When an node failure has caused a restart, we apply the same procedure as when we load a saved simulation (except for that another file is used).

### 6.3.9 Graphical debugger

As a help in developing the system, we wrote a graphical debugger consisting of three windows in Tcl/Tk [16].

The main debugging window displays written output from the processes (see Figure 6.4). Different system components can be observed simultaneously, or one by one (the components are the names in the toggle-buttons to the right in the figure). If one component is chosen, output from all processes that are instances of that component is displayed. A single process’ output can not be monitored in this window\(^6\). The window also provides commands for some of the services the Master module offers.

The process window (see Figure 6.5) displays the nodes, and processes on selected nodes. The white icons represent processes, and the work-station icons represent nodes in the system. Displaying the processes on a node is done by clicking on the desired icon. Processes can be displayed a node at a time, or several nodes simultaneously. Migration of a process is done by a drag-and-drop movement of the process icon to the icon representing the desired machine.

The third window (see Figure 6.6) displays statistics for the nodes, in terms of past load (number of reductions) future load (length of run-queue), Unix load and a total weight of the node load.

\(^6\)Unless, of course, the module only has one process instance.
Figure 6.4: Example screen for main debugging window

Figure 6.5: Example screen for viewing processes
Figure 6.6: Example screen for statistics
6.4 Conclusions

6.4.1 Declarative programming is nice

ERLANG proved to be a very good language for building prototypes fast. This work has taken six weeks (although the work has been prepared during 20 weeks), which was six stimulating weeks, since one could see the progress being made quickly.
Chapter 7

Clients

by Key Hyckenberg, Alexander Bottema, Per Jonsson, Pekka Hedqvist, Martin Argenius, Nils Meinhard, Terje Lundin

7.1 Introduction

The sim94 project is intended for training of order-giving of military personal, by concurrent simulation of troops in a dynamic terrain. The simulation consists of autonomous, communicating processes, implemented in the high-level language Erlang[7].

This article is devoted to the client part of the system. Clients connect to the simulation through a Unix port and are provided an interface to view the simulation map, troops and their movements, to send orders and to see battles as they happen. A client can be of two types, user or superuser, this is determined at login via the interface to the simulation.

Simulation-users are the persons who will receive training through interacting with the simulation. Superusers are the persons who supervise the users training. A user is always associated with a nationality Agents/Nationality2.6 process within the simulation. The user can view and send orders to its own troops. It can also see other nationalities troops and changes in the map which are detected by its own troops.

A superuser can choose, from several views, any of the current nationalities. Or the superuser can view the simulation as the “world” process, which views all troops and map changes of the simulation. The superuser can send orders to any troop, view the status of any troop, create and destroy troops, set areas on the map to forbidden or reset them to ”open terrain” (to simulate contaminated areas). The interface is the same for both users and superuser but some functions are not enabled for normal users.

We emphasise that superusers are not simulation system administrators; they are the game supervisors and need no particular computer knowledge. System specific parts, such as process migration, are done in the master module.

The user-interface is implemented in STk[11, 16] (Scheme TK). It is separated from the Erlang part and implemented as a client which communicates
with the simulation through a Unix port. Hence, users and superusers can be located anywhere where TCP/IP[13] networking is available. Any number of clients can connect/disconnect to the simulation at any time. The simulation is not dependent on connected clients, and can, when properly setup, run without them.

Every client connected to the simulation has a corresponding tokenizer process inside the simulation which handles the communication with the client. A communication protocolB has been specified and implemented for this part. The client is thus simulation, language and system independent, and can therefore easily, provided the protocol, be rewritten in another programming language on another platform.

We choose STk because we needed a free and competent X toolkit and a nice programming language. The decision of STk rather than Tcl/Tk was based upon the assumption that a higher level functional language is better suited for large applications. Our previous experiences of Tcl/Tk showed that it was hard to maintain good code quality for larger applications. And previous experiences of functional languages made us choose Scheme in favour of Tcl. In order to simplify and speed up development some extensions of the original STk source code were added.

7.2 Preliminaries

In implementing the client module an extended version of STk were used for the user-interface. STk stands for Scheme[11] Tk[16]. Scheme is a functional (declarative) programming language highly inspired by LISP[15]. Tk is an X[14] widget toolkit inspired in look and feel by Motif[12]. STk is a Scheme with Tk highly integrated in the language.

The client communicates with the simulation through a Unix port. Every client has a corresponding tokenizer within the simulation to handle the communication. The tokenizer consists internally of two processes, one for requests from the client, one for answers from the simulation. The tokenizer implements a specified communication protocol between the client and the simulation. Internally the tokenizer is a simple LR[5] parser. When the tokenizer receives requests from the user interface it translates them to corresponding Erlang messages. The tokenizer and the buffer also need means to break down Erlang messages to byte lists before the byte lists are sent to the socket and further to the “outside”. It uses a function, pack, also referred to as the packer. The packer is basically the inverse of the parser in the tokenizer.

The asynchronous communication from the simulator is based on events[10]. The events are generated from the simulator and handled by the buffer.

All functions the tokenizer calls are implemented as Remote Procedure Calls (RPC[9]). Remote Procedure Calls are synchronous messages to other processes who syntactically looks like procedure calls. The result of the message is returned like in a normal procedure call and the sending (calling) process is halted until the answer is received. Troop orders are also imple-
mented as RPC calls but the calls only returns a dummy value. Each order is provided with a order reference so that the event generated when the order is done or considered as impossible can be identified. That is because we like to be able to give many orders to different troops at once without waiting for the answer of every troop if the order could be performed.

7.3 The user Module

The user interface consists of an “inside” written in Erlang that directly communicates with the processes of the simulator and an “outside” currently written in STk.

7.3.1 Erlang system dependent

The “inside” of the user interface consists of two separate processes, the tokenizer and the buffer. The buffer reads data from the socket and receives events from the simulation. The buffer is the only process the rest of the simulator is aware of. The tokenizer interprets messages from the user and takes appropriate action. It is also hidden from the rest of the simulator and is only an implementation issue. The reason for having one process for reading and parsing input from the socket is that time consuming RPC’s should not block outgoing events. When no ambiguity can arise the term tokenizer will be used for the pair of tokenizer and buffer processes. When other processes refers to the tokenizer they refer to the tokenizer-buffer pair.

When a user connects to the simulator through a socket a buffer is spawned which in turn spawns a tokenizer. Together the buffer process and the tokenizer process constitute the tokenizing facility. The socket server spawned by the master module sends data as Erlang messages to the buffer which the tokenizer reads. The tokenizer process parses the input and makes the appropriate function call. Events are sent to the buffer which packs the event depending data sent with the event and sends the packed data as a byte list to the socket server.

The tokenizer process parses the received messages from the socket server. The messages consists of a tag and a byte-list. When the tokenizer is ready for input it tries to read from the buffer process. The byte list read is parsed and the tokenizer calls the appropriate Erlang function. The call is synchronous but the reply may be ignored. If the return value of the function is to be returned to the client the value is converted to a byte list and sent to the socket. If the function only sets some options in the simulator and the reply is ignored by the interface nothing will happen. The tokenizer is now ready for a new input.

The only process id belonging to the socket communication the simulator knows about is the buffer process id. All explicit communication from within the system (not RPC’s) to the client is via the buffer.

Events are handled by the buffer process. When an event is generated from the simulator a message is sent to the buffer. The message sent to the buffer contains a tag specifying what kind of event it is, e.g an agent
has died, the process id of the generating process and some event depending data that can be examined. The data is packed with the packer, given a tag specifying what kind of event it is, and sent to the socket.

![Diagram of the topology of the tokenizer-buffer communication](image)

**Figure 7.1:** The topology of the tokenizer-buffer communication

### 7.3.2 Scheme language dependent

As previously mentioned, we chose STk as the tool language for the graphical user interface. Since the standard release of STk did not provide socket communication it was explicitly added. Hence, the name Sim94STk.

There were mainly two major changes; the io-interface was rewritten so that events were generated whenever incoming data are detected at a data stream and Scheme interface primitives were provided to setup initialisation of a socket handler. The socket handler is simply any function which is called upon received data on the socket. Other changes were quite minor, such as making the io-interface 8-bit instead of the standard 7-bit (which is useful for raw data socket communication).

The map consists of more than 20000 line vectors and memory trashing became a serious problem when the map initialiser was written in Scheme. Hence, we added an STk primitive read-map that reads the map and issues Tk events directly from C, thus avoiding the problems with big data structures in Scheme. As a result, the map initialisation became almost 100 times faster.

### 7.4 Implementation outline

The following algorithms, data structures and implementation techniques have been used.

The aim for the implementation of the user interface was to enable the use of this simulator in a heterogeneous network. The user interface does not have to be written in Erlang. Another aim was that a user should be able to connect and disconnect without interfering with the simulator. To achieve the goals we use socket communication over TCP/IP. In Erlang there are built-in functions for socket communication. The master module spawns a socket server and allocates a socket. The socket server handles all communication via the socket from all connected clients, i.e. there is only one socket server in the simulator. When no ambiguity can arise reading
from the socket will be considered as equivalent to receiving a message from
the socket server.

7.4.1 Erlang system dependent

The parser used in the tokenizer to interpret the byte lists is a simple LR
parser.

The parser reads four bytes from the buffer (if possible) and interprets
them as a 32 bit integer sent most significant byte first. The integer denotes
some action in the simulator. If the action needs some arguments the argu-
ments was to be sent at the same time over the socket and the arguments
should reside in the buffer. The arguments are read from the buffer and
the appropriate action is taken or appropriate function call is made. The
arguments are structured according to the protocolB.

Some functions do not return a well defined value but puts the simulation
in a different state.

It is needed to have a function that converts a Erlang message to a byte
list, a packer. The packer takes a tag and a Erlang term and returns a list
of bytes. The tag specifies what kind of Erlang term is to be packed. The
packer is basically the inverse of the parser but since some kind of messages
will only be sent in one direction parser and packer are only partial inverses
of each other.

The buffer uses a simple queue from which the tokenizer tries to read by
dequeueing bytes from it. If the buffer is empty the tokenizer will wait until
the buffer receives data from the socket. The buffer never interprets the
data received from the socket server and always interprets the data received
from the simulator.

Notice that even if the tokenizer hangs the buffer can still receive events
and data from the socket.

The robustness of the tokenizer-buffer is achieved by using a catch on
function calls. If a function call fails or something unexpected happens the
call will return some default value instead of terminate. If the reply does
not look like expected the reply is thrown away and not sent to the user
interface. The design of the Scheme part of the interface with events as the
model of communication enables us to throw away the messages without
deadlock occurring.

The buffer/tokenizer processes are linked Erlang processes. If one of the
buffer/tokenizer processes terminates from any reason the other will also
terminate immediately. This certifies that that they can be seen as one
computing facility from the master module point of view and there will be
no “leakage” of processes.

If corrupt data is received the buffer will be emptied and the request will
be lost. No warning to the connected client will be issued, but a warning
message is written in the debug window Master/Graphical Debugger6.3.9.
7.4.2 STk language dependent

Tk (ToolKit) is a tool language to setup and initialise the actual graphical objects, widgets, on the screen. It is quite easy to specify the layout and contents of the menus, windows and input entries. Each widget is then bound to some lambda abstraction which is called upon user interaction, e.g. whenever the user clicks with the mouse on a Tk widget such as a button. As perhaps familiar, this is the essence of “event driven programming”; widgets are bound to events and related functions so that whenever an event has occurred the corresponding function is called for that particular widget.

Graphical update of the simulation is based upon the information obtained by the socket port. A simple protocol is specified for all types of events that occur in the simulation. All communication was made asynchronous to prevent interface lockups. Once events are received on the socket port they become decoded and the appropriate functions are called to maintain a consistent view of the simulation. For instance, there are events for newly arrived units, destroyed units, units entering battle and changes of the terrain. See the complete protocolB for actual details.

GUI Implementation Issues

Currently the interface consists of 14 separate modules (or files) that are loaded upon boot time. Briefly the modules are:

- **map.**STk Map related functions. Functions that provide map window initialisation, rescaling and postscript printing. Routines for translating mouse cursor position to map coordinates are also provided.
- **windows.**STk Routines to display various pop-up windows, e.g. the login window, getting host and socket port.
- **commands.**STk Routines to handle various commands provided by the tool-box including tool-box window initialisation. Currently the tool-box provides commands for unit assertion, unit deletion, order issuing, unit status and current unit orders.
- **menus.**STk Setup and initialise menus. Current menus are File, Commands, Map and Views.
- **socket-io.**STk Socket IO functions, i.e. functions to connect and initialise socket and to (un)marshall various type of data (integers or character strings). The socket connection is opened as a raw data stream connection using 8-bit characters as the smallest discrete units. The top level function for receiving events is also provided here. Once an event has been identified, further interpretation is done in a separate function of the “replies” module.
- **units.**STk Routines to handle units (troops) displayed on map. Functions to add units, remove units or move units are called once corresponding events are received on the socket connection. An abstract
data type is provided for keeping track of all visible units. Currently this structure is a list. Each unit has an identification number (this is unique even between nationalities) a graphical Tk-object reference (i.e. its graphical representation), name (user defined), side (to which nationality the unit belongs) and its position.

• views.STk Each nationality is provided with a separate view of the world. This is because separate nationalities may own private knowledge about the world, e.g. where certain enemies are visible. This information is not necessarily consistent with the actual representation of the world. An abstract data type is provided to keep track of all available views. These are updated incrementally whenever such events are entered on the socket connection. It is also possible to update the entire list of current views if it is issued to the simulation server. Note that there are no local information at all in the graphical user interface. All information is stored in the simulation server and the user interface updates its information upon events from the socket connection.

• error-msg.STk Error messages. If something goes wrong, e.g. the socket connection promptly goes down, the user is immediately informed.

• replies.STk Once events from the socket connection have been identified the appropriate function is called in this module for further interpretation. For example, when a unit assertion event has been identified, a function is called in replies that interprets the resulting parts of the event such as identifier, name, position and nationality. All received events are interpreted in this module.

• tool-win.STk Separate tools from the tool-box provide various parameters. These are displayed in a window beneath the tool-box and is immediately updated when a particular tool is selected. This module contains functions to initialise such windows.

• area.STk An abstract data type is provided to keep track of all visible areas, i.e. modified terrain. They are stored in a list and functions to add, remove, redisplay, move areas are provided.

• states.STk An abstract data type is provided to keep track of all saved states. Briefly, this is just a plain list of names. A new state is loaded by issuing a load request with a state name to the simulation server.

• misc.STk Miscellaneous functions.

• main.STk Boot and initialisation module.

Recall that there is no information stored locally. This is to prevent loss of information if the local computer where the client is running cease to
work. It should also be possible to log on to the simulation server, logout
and log on later to continue to interact with the simulation.

Based upon these requirements we have to use a protocol where any
information can be retrieved and stored. Furthermore, this protocol should
be entirely asynchronous to prevent interface deadlocks. However, this also
burdens the programmer with synchronisation issues. In some cases this can
be very troublesome, e.g. suppose that the (super)user suddenly switches
view meanwhile the server updates the units of the current view. Then the
position of the troops in the old view is sent to the user interface, but the
user does not want to see them. The user will not receive any updates from
the old view but the troops are still visible on the screen together with the
ones in the new view. The user has to do a “refresh” to clear the screen
from the old troops.

7.5 Conclusions

We have concluded that Erlang is an appropriate language for process ori-
eted programming. The development speed was truly significant due to
the high-level nature provided by Erlang; convenient data structures (lists
and tuples) and garbage collection prevented us from memory leakage and
having to consider irrelevant details.

Even though STk is far better than Tcl/Tk we have had some problems.
The first problem is that STk locks the interpretation of events while pro-
cessing a function; there are no lightweight threads that one can use. The
second problem is that one must store the information globally to memorise
events. Hence, the functional paradigm is highly different from the style of
event driven programming. Since Scheme after all is a functional language,
a paradigm incompatibility arises. Thus, according to our own experiences,
the search for a convenient graphical tool language remains unsolved. A pos-
sible solution could be to merge Erlang with Tk; Erlang has both lightweight
processes and the nice feature to store data in processes without destroying
the essence of functional programming.

7.6 Examples

In figure 7.2 the graphical user interface as seen by a superuser is shown.
The difference for an non superuser is that some of the tools in the tool-box
in the upper right corner is not available for him.

In figure 7.3 the tool-box is shown in close up. A show info of a specific
troop has been issued. The fields shows the available information of that
troop.

In figure 7.4 the user can see how a troop discovers a forbidden area.
Only the Area within seeing range is discovered to be forbidden. The troop
now has to inform the navigator and wait for a new path.

In figure 7.5 the path of a troop is showed. The path is shown if the user
chooses the show info tool in the tool-box.
In figure 7.6 the dialogue window for removing orders is shown. The current order is in the bottom of the list. Each order consists of a reference, an action, a coordinate, begin time and end time. The time 0 0.0:0 means ASAP.

Figure 7.2: The graphical user interface.
1. Troop symbol is used to place new troops
2. Death-skull removes troops
3. Square is used to mark open terrain or forbidden area
4. Crossed square is used to undo area markings
5. Expression mark is used to give troop orders
6. Question mark gives info about selected troop
7. Expr.mark+arrow retracts order for selected troop
8. Foo - only there for the pleasure of pressing buttons

Figure 7.3: The tool-box buttons from left to right
Figure 7.4: If a troop reaches a forbidden area it stops. The area in its view is revealed.

Figure 7.5: Here you can see how a troop finds its way to the desired location.
Figure 7.6: The dialogue window for removing orders.
Chapter 8

Conclusion

We have presented a process-based implementation of a concurrent simulator for plan-driven troops. A process-based design is obviously a good choice for this problem, but is a process-based implementation efficient enough? Our simulator has verified that the implementation technique is efficient enough, although one can certainly find many ways to improve this first prototype.

The short development time, seven weeks, also proves that the use of high-level tools like ERLANG and SchemeTk drastically increases the productivity.
Bibliography


Appendix A

View

View is a one-way interface through which User/Client can get information about the running simulation. In the simulation there is a set of Views, each View corresponding to a process acting as a View Provider. Such a process should conform to the View interface and be able to answer queries defined by the interface. Currently the only View Providers are the World process and the Nationality processes.

Figure A.1: The design of the view interface

Figure A.1 shows how the View Interface is constructed. Client processes use the functions defined in the View module to retrieve information from a View Provider. The dots in the View Module can be seen as entry points for the functions in the module. Depending on the View identifier supplied with the call, the request is passed to the corresponding View Provider.

Example A.0.1 To get all known agents from a specified View, the client makes use of the following function of the View module:

AllAgents = view:getAgents(MyViewId),
In addition to the functional approach, as illustrated in example A.0.1, the View interface introduces the notion of subscriptions. Their functionality is analogous to that of the real world, and thus provides a way for the Client to asynchronously get notified when events occur within a specified View Provider.

The notification is done by a message sent from the View Provider to the subscribing process. This message is called a notification. Example A.0.2, shows a possible use of subscriptions.

**Example A.0.2** By issuing the following code fragment once, the Client associated with UserId will get notified every time an agent’s death becomes known to the View Provider identified by the ViewId.

```
view:subscribe_event(ViewId, UserId, agent_died),
```

The intention with the View interface is to allow Users to play different roles in the simulation. By connecting to a View provided by a Nationality for example, the information retrieved will be based on the knowledge of that nationality. Facts about enemy agents will be sparse, if present at all, depending on the amount of knowledge this nationality has of these enemy agents.

On the other hand, by connecting to the View provided by the World process, the Client takes on the role of game supervisor, thus getting the truth, the whole truth and nothing but the truth :) World has the correct status of all agents, as well as the current and complete properties of the terrain.

The set of possible Views is retrieved by the Client from the Master process, either during initialisation, or on the request of changing view from the user. This makes the information retrieval from the simulation quite dynamic.
Appendix B

Socket communication protocol

by Alexander Jean-Claude Bottema, Per Jonsson

B.1 The Protocol

Data are divided into various categories that are represented by streams of tokens. Each token is either a 32 bit integer or a single character (8-bit integer). The semantics of each datum is determined by the first token as an “opcode” identifier.

Commands (that are issued from client to tokenizer) are identified with opcode numbers less than 1000. Corresponding replies/acknowledgements are the same opcode numbers with an offset of 1000, e.g. IssueOrder has opcode 1 and OrderReply correspondingly 1001, etc.

Furthermore, we will implicitly assume that:

- denotes 32 bit integers.
- denotes character sequences (with a fix number of characters).

Named fields in records are denoted as:

token id = field name
### B.2 Client $\to$ Tokenizer

#### B.2.1 Order Issuing

<table>
<thead>
<tr>
<th>Order Type</th>
<th>AgentId</th>
<th>Reference</th>
<th>OrderType</th>
</tr>
</thead>
</table>

1 = Take  
2 = Defend  
3 = Reorganize

<table>
<thead>
<tr>
<th>Start time</th>
<th>Finish time</th>
<th>Dest. X-coord</th>
<th>Dest. Y-coord</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Days</th>
<th>Hours</th>
<th>Minutes</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Days</td>
<td>Hours</td>
<td>Minutes</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

#### B.2.2 Create Agent

<table>
<thead>
<tr>
<th>Number of fields in record</th>
<th>Name</th>
<th>length in chars</th>
<th>Nationality</th>
<th>32 bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Position</th>
<th>32 bit value</th>
<th>Type</th>
<th>32 bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>32 bit value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Man power</th>
<th>32 bit value</th>
<th>Weapon range</th>
<th>32 bit value</th>
<th>Fire power</th>
<th>32 bit value</th>
<th>Ammunition</th>
<th>32 bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
B.2.3 Kill Agent

3
AgentId

B.2.4 Create Nationality

4
Name
  length in chars
Password
  length in chars

B.2.5 Kill Nationality

5
NationalityId

B.2.6 Get all Views

6

B.2.7 Get Agents

7
ViewId

B.2.8 Subscribe Movements

8
ViewId

B.2.9 Subscribe Map Events

9
ViewId

B.2.10 Subscribe Agent Battle

10
ViewId

B.2.11 Subscribe New Units

11
ViewId

B.2.12 Subscribe Killed Units

12
ViewId
### B.2.13 Unsubscribe Movements

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>ViewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td></td>
<td>ViewId</td>
</tr>
</tbody>
</table>

### B.2.14 Unsubscribe Map Events

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>ViewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td></td>
<td>ViewId</td>
</tr>
</tbody>
</table>

### B.2.15 Unsubscribe Agent Battle

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>ViewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
<td>ViewId</td>
</tr>
</tbody>
</table>

### B.2.16 Unsubscribe New Units

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>ViewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
<td>ViewId</td>
</tr>
</tbody>
</table>

### B.2.17 Unsubscribe Killed Units

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>ViewId</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td></td>
<td>ViewId</td>
</tr>
</tbody>
</table>

### B.2.18 Cancel Order

<table>
<thead>
<tr>
<th></th>
<th>Agent Id</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.2.19 Set Area

<table>
<thead>
<tr>
<th></th>
<th>TerrainType</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left X-coord</td>
<td>0=Open terrain</td>
</tr>
<tr>
<td>Upper Y-coord</td>
<td>5=Forbidden</td>
</tr>
<tr>
<td>Right X-coord</td>
<td></td>
</tr>
<tr>
<td>Lower Y-coord</td>
<td></td>
</tr>
</tbody>
</table>
B.2.20  Remove Area

20

Left X-coord
Upper Y-coord
Right X-coord
Lower Y-coord

B.2.21  Get all Areas

21 ViewId

B.2.22  Get Agent Status

22 AgentId

B.2.23  Get Path of Agent

24 AgentId

B.2.24  Login

25 NationalityId

Password  length in chars  \[\oplus\] \[\oplus\] \[\oplus\] \[\textbullet\]

B.2.25  Set Password

26 NationalityId

New password  length in chars  \[\oplus\] \[\oplus\] \[\oplus\] \[\textbullet\]
Monitor password  length in chars  \[\oplus\] \[\oplus\] \[\textbullet\]

B.2.26  Set Time Intervals

27

Simulated time

Days Hours

Minutes Seconds
<table>
<thead>
<tr>
<th>B.2.27</th>
<th>Load State</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Name of state</td>
<td>length in chars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B.2.28</th>
<th>Save Current State</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Name of state</td>
<td>length in chars</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B.2.29</th>
<th>Get Saved States</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B.2.30</th>
<th>Get Orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>AgentId</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B.2.31</th>
<th>Remove order</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>AgentId</td>
</tr>
<tr>
<td></td>
<td>OrderRef</td>
</tr>
</tbody>
</table>
### B.3 Client ↔ Tokenizer

#### B.3.1 Order Reply

<table>
<thead>
<tr>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentId</td>
</tr>
<tr>
<td>Message</td>
</tr>
</tbody>
</table>

#### B.3.2 Create Nationality Reply

<table>
<thead>
<tr>
<th>1004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>NationalityId</td>
</tr>
<tr>
<td>ViewId</td>
</tr>
</tbody>
</table>

#### B.3.3 Get all Views Reply

<table>
<thead>
<tr>
<th>1006</th>
<th>No. of triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewId</td>
<td>NationalityId</td>
</tr>
<tr>
<td>Name of Nationality</td>
<td>length in chars</td>
</tr>
</tbody>
</table>

#### B.3.4 Get Agents Reply

<table>
<thead>
<tr>
<th>1007</th>
<th>No. of agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AgentId</td>
<td></td>
</tr>
<tr>
<td>Number of fields in record</td>
<td></td>
</tr>
<tr>
<td>1 = Name</td>
<td>length in chars</td>
</tr>
<tr>
<td>2 = Nationality</td>
<td>32 bit value</td>
</tr>
<tr>
<td>3 = Position</td>
<td>X</td>
</tr>
<tr>
<td>AgentId</td>
<td></td>
</tr>
</tbody>
</table>

...
### B.3.5 Movement Events

<table>
<thead>
<tr>
<th>1008</th>
<th>AgentId</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X value</td>
</tr>
<tr>
<td></td>
<td>Y value</td>
</tr>
</tbody>
</table>

### B.3.6 Map Area Events

<table>
<thead>
<tr>
<th>1009</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-1=Remove area</td>
</tr>
<tr>
<td></td>
<td>0=Open terrain</td>
</tr>
<tr>
<td></td>
<td>5=Forbidden</td>
</tr>
</tbody>
</table>

### B.3.7 Agent Battle

<table>
<thead>
<tr>
<th>1010</th>
<th>AgentId</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Start or Stop</td>
</tr>
<tr>
<td></td>
<td>0=Starting</td>
</tr>
<tr>
<td></td>
<td>1=Stopping</td>
</tr>
</tbody>
</table>

### B.3.8 New Unit Events

<table>
<thead>
<tr>
<th>1011</th>
<th>AgentId</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 = Name</td>
</tr>
<tr>
<td></td>
<td>length in chars</td>
</tr>
<tr>
<td></td>
<td>2 = Nationality</td>
</tr>
<tr>
<td></td>
<td>32 bit value</td>
</tr>
<tr>
<td></td>
<td>3 = Position</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

### B.3.9 Killed Unit Events

| 1012 | AgentId |
### B.3.10 Get Areas Reply

<table>
<thead>
<tr>
<th>No. of areas</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1021</td>
<td>0=Open terrain 5=Forbidden</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Left X-coord</th>
<th>Upper Y-coord</th>
<th>Right X-coord</th>
<th>Lower Y-coord</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.3.11 Get Agent Status Reply

<table>
<thead>
<tr>
<th>AgentId</th>
<th>Number of fields in record</th>
<th>Length in chars</th>
<th>32 bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1022</td>
<td>1 = Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 = Nationality</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 = Position</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>4 = Type</td>
<td>32 bit value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 = Man power</td>
<td>32 bit value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 = Weapon range</td>
<td>32 bit value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 = Fire power</td>
<td>32 bit value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 = Ammunition</td>
<td>32 bit value</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9 = Action</td>
<td>length in chars</td>
<td></td>
</tr>
</tbody>
</table>

### B.3.12 Time Event

<table>
<thead>
<tr>
<th>AgentId</th>
<th>Time Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1023</td>
<td></td>
</tr>
</tbody>
</table>
### 3.13 Get Path of Agent Reply

<table>
<thead>
<tr>
<th>Days</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

1024

\[ n = \text{Number of points (possibly zero)} \]

| \( X_0 \) | \( Y_0 \) |
| \( X_1 \) | \( Y_1 \) |
| \vdots | \vdots |
| \( X_{n-1} \) | \( Y_{n-1} \) |

### 3.14 Login Reply

1025

Authority

0 = Wrong passwd
1 = Correct passwd

### 3.15 Set Password Reply

1026

Authority

0 = Wrong passwd
1 = Correct passwd

### 3.16 Load State Reply

1028

Authority

0 = Unsuccessful load
1 = Successful load

### 3.17 Save State Reply

1029

Authority

0 = Unsuccessful save
1 = Successful save
### B.3.18 Get Saved States Reply

<table>
<thead>
<tr>
<th>State 0</th>
<th>length in chars</th>
<th>[ \cdots ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>State 1</td>
<td>length in chars</td>
<td>[ \cdots ]</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>State (n-1)</td>
<td>length in chars</td>
<td>[ \cdots ]</td>
</tr>
</tbody>
</table>

### B.3.19 Simulation restated

<table>
<thead>
<tr>
<th>1031</th>
</tr>
</thead>
</table>

### B.3.20 Get orders reply

<table>
<thead>
<tr>
<th>1035</th>
<th>Number of Orders</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>AgentId</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OrderType</th>
</tr>
</thead>
</table>

| 1=Take |
| 2=Defend |
| 3=Reorganize |

<table>
<thead>
<tr>
<th>Start time</th>
<th>Days</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>Seconds</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Finish time</th>
<th>Days</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minutes</td>
<td>Seconds</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dest. X-coord</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Dest. Y-coord</th>
</tr>
</thead>
</table>