An Experiment In Automatic Adaptor Generation by means of Evolution

Werner Van Belle
Werner.Van.Belle@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

http://prog.vub.ac.be/CBorg
Vrije Universiteit Brussel - Programming Technology Lab
Pleinlaan 2, 1050 Brussel - Belgium

Abstract

Mobile multi agent systems can be seen as a basis for global peer to peer computing. Nevertheless such an open environment makes it difficult to write software which can interface with other agents and is an excellent area for interface conflicts. Conflicting interfaces can be remedied by using adaptors. We show that it is possible to automatically generate those glue-adaptors. We do this by means of genetically engineered classifier systems, which use Petri-nets as a model for the underlying interfaces. In this paper we report on such an experiment.

1 Introduction

Mobile multi agent systems are a good basis for global peer to peer computing. In such a world, agents communicate with each other to reach specific goals. Applications are defined by the interactions of multiple agents which usally are written by different developers and different organisations. This forms our problem: every developer which needs another agent relies upon that agents' API. If this API, or the semantics of the other agent's API change slightly, the calling agent will not work anymore. This will eventually result in a cascaded application breakdown. As long as we don't solve the problem of interface differentiation in some automatic fashion, mobile multi agent systems may just be too impractical for programmers to use.

This paper reports on an experiment we performed to create automatic interface adaptors between agents. First we will explain what view we use on mobile multi agent systems, afterwards we explain what kind of typical interface conflict we use in our experiment. Then we explain how we use Petri-nets to model each interface, which leads us to section 5 where we discuss the genetic programming technique we used to generate an adaption interface automaticaly. Section 6 then shows how this works on our previously introduced case.

2 Mobile Multi Agent Systems

A mobile multi-agent is an active autonomous software component that is able to communicate with other agents; the term mobile refers to the fact that an agent can migrate to other agent systems, thereby carrying its program code and data along with itself. Regarding the terminology *mobile multi-agent system* [BFDV01]there is some confusion. A multi-agent system in AI denotes a software system that simulates the behavior of large groups of interacting agents (called multi-agents), without focusing on the distribution aspect of these systems. In the world of distributed computing, a mobile multi-agent system is a *distributed* environment in which multi-agents can be written. We adopt this second definition.

Mobile multi agents can be compared to processes. Agents can communicate with each other only by sending messages over a communication channel [BU00]. An agent can have multiple communication channels: one for every communication partner, or more than one for every interface on a communication partner. Communication channels are offered by means of ports on the agent.

Agents communicate asynchronously and always copy their messages completely upon sending. This, together with the fact that we work connection-based, allows us to place adaptors in between two agents.

3 The Case

The case we will use in our experiment is a typical problem of open distributed systems: how agents synchronise with each-other. In closed distributed systems this is not really a problem. A server offers a concurrency interface (typically a transaction interface) which can be used by clients. The clients have to adhere to this specification or they won't work. Since the server concurrency API doesn't change that much this is a usable setup.

In peer to peer computing, every agent has to offer a concurrency interface, and has to use the concurrency interfaces provided by other agents. To which extent an agent offers a concurrency interface is often a problem[Lea00]. It can offer a full fledged optimal transaction interface or it can offer a simple lock/unlock interface.

In our example we use a simple lock/unlock interface which a client can typically use to lock a resource and then use it. This API is as follows:

```
port concurrency
incoming lock(resource)
outgoing lock_true(resource)
outgoing lock_false(resource)
  // lock_true and lock_false are sent back whenever a lock
  // request comes in: lock_true when the resource is locked,
  // lock_false when the resource couldn't be locked.
incoming unlock(resource)
outgoing unlock_done(resource)
  // will unlock the resource. Send unlock_done back when done.
```

In our case, the agent also offers some behaviour through another connection. In our example we will keep this behaviour as simple as possible.

```
port behaviour
incoming act(resource)
outgoing act_done(resource)
  // will do some action on the agent.
```

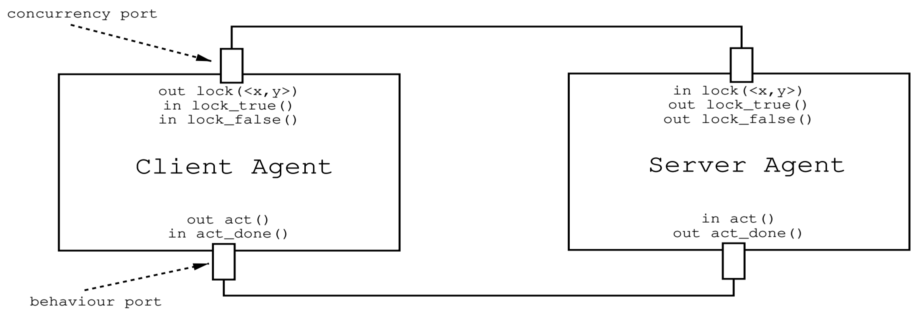


Figure 1. Client and server agent, both honouring the same interface. Agents interact with each other over ports. This enables us to easily place adaptors in between them.

Even with this simple concurrency interface the semantics of the interface can be slightly different. We will use two kinds of concurrency interfaces:

Nested Locking semantics In our first version, somebody can lock a resource multiple times. Every time the resource is locked the lockcounter is increased. If the resource is unlocked the lockcoutner is decreased. The resource is finally unlocked when this counter reaches zero. These semantics allows us to use routines which autonomously lock resources.

Non-Nested Locking semantics The second version of our locking semantics, doesn't offer a counter. It simply remembers who has locked a resource and doesn't allow a second lock. When unlocked, the resource is fully unlocked.

Differences in how the API consideres 'lock' and 'unlock' makes agents fail or succeed. In our next example we will illustrate how we can model such differences by means of Petri-nets

4 Petri-nets

A Petri-net is a formal model for concurrent systems [Rei]which is extremely suitable for our case. Petri-nets offer a model in which we can have multiple actors which each go from state to state by means of state transitions. Both concepts: the states as well as the state transitions, are modelled.

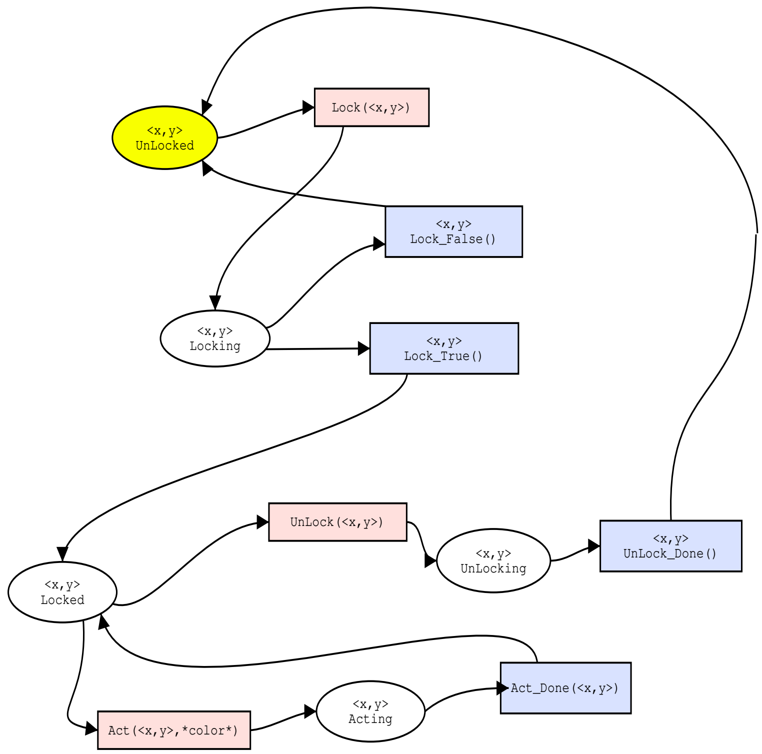


Figure: Non Nested locking strategy, described in a Petri-net. The red boxes are the messages which either comes in are goes out. The blue ones are exactly the opposite.

For example, in the above figure we see how a non-nested locking strategy is modelled. States are shown as circles, while transitions are shown as boxes. The current state (*unlocked*) is colored in yellow. From this state we can choose only one action: *lock*. When we choose this action, we are in the locking state until *lock_true* or *lock_false* is performed. Note that we can view this model from two angles. First, it can specify what the server offers to clients. Second it can specify what a client expects from the server. In other words, this model does not specify whether a client requires or provides an interface. It is perfectly possible to offer an interface which adheres to this specification, in which case, the incoming *lock* is initiated from the client, and *lock_true* or *lock_false* are sent back to the client when making the transition. If this interface is required, the *lock* transition is made when we send a lock request to the server, and a *lock_true* or *lock_false* transition is made when the server responds.

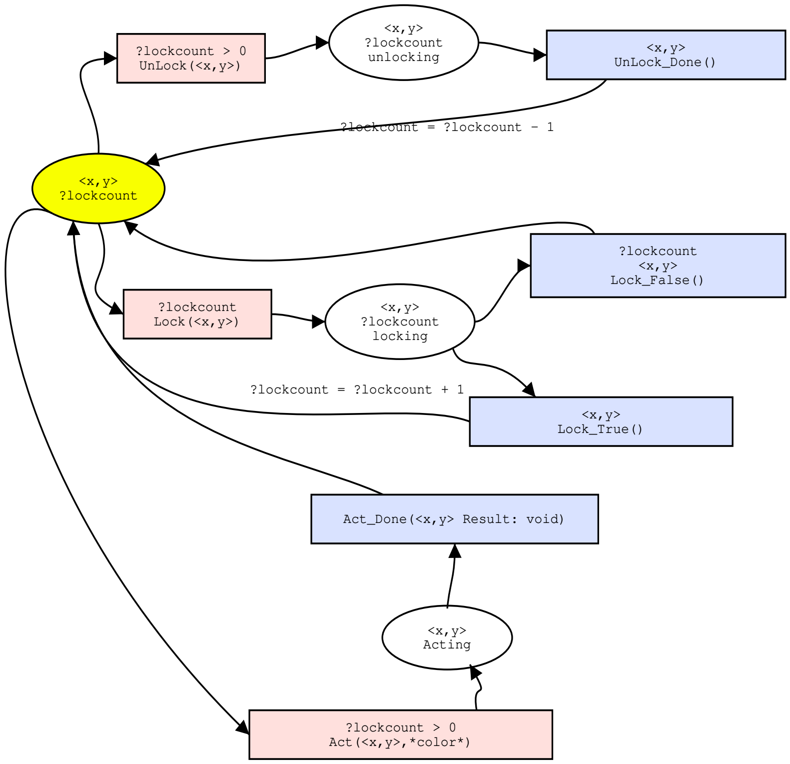


Figure: Nested Locking Strategy described as a Petri-net.

In the above figure we see how a transition can only be made when a certain condition holds true: We can only *unlock* when we have a lockcount which is larger than 0. The state holds this extra field: ?*lockcount*.

Now, we will use these Petri-nets to model the behaviour of server and client. More importantly this tool helps us to model the possible actions in a certain context, which is defined partly by the client and partly by the server.

5 Classifier Systems

A classifier system [Pol96] is a system which has an input interface, a classifier-list and an output interface. The input and output interface put and get messages to and from the classifier-list. The classifierlist takes messages as input and produces new messges as output. Every message is a set of bits (only 0 or 1 :) which are matched against a list of classifiers. A classifier is a condition/action pair.[1]

Conditions and actions are both strings which can contain 0, 1 or #. A '#'-character is a pass-through which, in a condition, means 'either 0 or 1 matches'. If found in an action, we simply replace that character with the character from the original message. Consider for example the following table:

[1]For the sake of simplicity, we currently only consider classifiers with only one condition and without negation. If we want Turing completeness we should use at least two conditions and allow the negation of conditions. Furthermore we need a loop-back device in our output/input loop which can be modified by the classifiers.

Message	Classifier Condition	Classifier Action	Matches	Result
001	00#	111	yes	111
101	00#	000	no	/
110	1##	###	yes	110
100	1##	1#0	yes	100

Such a classifier system can be used to express the actions which should be taken in a certain context. In our locking/non-locking example, we need to define how such a (part of a) Petri-net is mapped onto a message. We define the semantics of our messages as a 28 bit tuple, with the following semantics

Server Possible Actions

L	L#t	L#f	U	U#	A	A#

Client Possible Actions

L	L#t	L#f	U	U#	A	A#

Server Requested Action

L	L#t	L#f	U	U#	A	A#

Client Requested Action

L	L#t	L#f	U	U#	A	A#

Figure: L = lock; L#t = lock_true; L#f = lock_false
U =unlock; U# = unlock_done
A = act; A# = act_done

Simply translating a lock from client to server requires three rules. One which transforms the client *lock* request into a server-lock call and two which transform a server *lock return* into a client *lock return*.

	Condition1	Action	Comments
1.	1###### ####### ####### 1######	####### ####### 1###### 0######	If we can call lock upon the server & the client wants to call it
2.	1###### #1##### #1##### #######	####### ####### #0##### #1#####	If we can call lock_true upon the client & the server wants to call it
3.	1###### ##1#### ##1#### #######	####### ####### ##0#### ##1####	If we can call lock_false upon the client & the server wants to call it

Figure: The bit patterns from left to right and top to bottom: server possible actions, client possible actions, server requested actions, client requested actions.

Although this is a simple example, more difficult actions can be represented. Suppose we are in a situation where the client uses a nested-locking strategy and the server uses a non-nested locking strategy. In such a situation we don't want to send out the lock-request to the server if the lockcount is larger than zero. We can easily represent such a condition by rule 4 and 5.

	Condition1	Action	Comments
4.	1###### 1##0### ####### 1######	####### ####### 1###### 0######	If the client wants to lock and has no lock we send the lock to the server
5.	####### 1##1### ####### 1######	####### ####### ####### 01#####	If the client wants to lock but has already a lock we immediatelly send back an lock_true.

6 Genetic Algorithms

A genetic algorithm[Koz92] is an algorithm which evolves a set of genes over a number of generations. Each individuals' fitness is measured in a certain context. When the gene pool is measured we keep (reproduce) 10% of the best individuals. We throw away 10% of the worst individuals and replace them with crossovers from the 10% best group. The

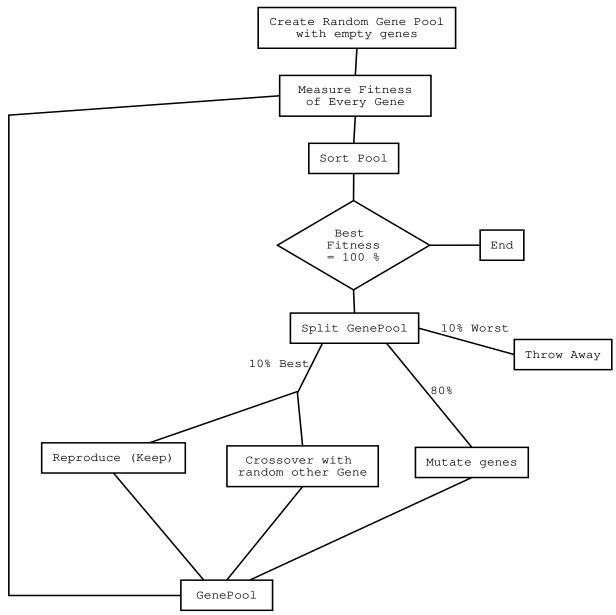


Figure 2. Flowchart of the conventional genetic algorithm

other 80% are simply mutated (randomly adapted). So, to make a genetic algorithm work we still need to define our individuals with crossover/mutation rules and a fitness measure.

Individuals We opted to use complete classifier-lists as individuals and let them compete with each other. Therefore, the classifier-lists are the genes of our genetic algorithm. This way of working is known as the Pittsburgh approach.

Enlarging Genes Genes are initially empty. Every time a certain situation arises and there is no matching rule we will create a new rule. This rule has a random condition with a random action on the server and/or the client from the list of possible actions. This guarantees that we don't start with completely useless rules which cover situations which will never exist and allows us to generate smaller genes which can be checked faster.

Mutating Genes When we mutate a gene, we select a new random action for each classifier.

Crossover Genes To create a crossover of genes we iterate over both classifier-lists and each time we throw a dice. If the result is true, we add the rule, if the result is false, we throw it away.

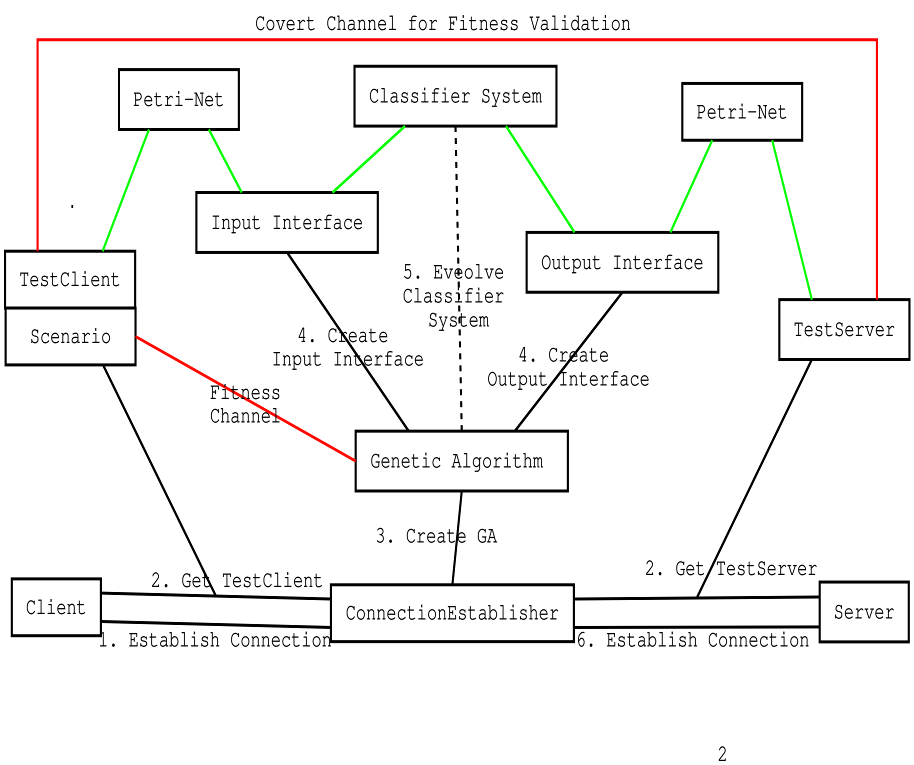
Measuring Fitness Measuring fitness is done by means of a number of test-sets (which will be discussed in the next section). Every test-set illustrates a typical behaviour (scenario) the client will request from the server. The fitness of a gene is determined by how far the scenario can execute without unexpected behaviour. Of course this is not enough, we should not have solutions which completely avoid the server. For example, return *lock_true* every time a request comes in from the client, without even contacting the server. To avoid this kind of behaviour from our algorithm the test-set will need a covert channel over which it can contact the server to verify some things.

7 The Experiment

7.1 Setup

The experiment is set up as a connection-broker between two components. The first component tries to make contact with the server by means of the broker. Before the broker sets up the connection it will generate an adaptor between the two parties to mediate slight semantic differences. It does this by requesting a test-agent from both parties. The client will client a test-client and a test-scenario. The server will produce a test-server. In comparison with the original agent, these testing agents has an extra *testing* port, over which we can reset them. Furthermore, this *testing* port is also used as the covert channel for validating actions at the server.

The genetic algorithm, using a gene pool of 100 agents, will now use the test-agents to measure the fitness of a particular classifier system. Only when a perfect solution is reached; ie a correct adaptor has been found, the connection is set up.



7.2 Scenarios

The scenarios offered by the client are the ones which will determine what kind of classifier system is generated. We have tried this with two scenarios. Scenario 1 is basically a sequence: [lock(), act(), unlock()] x 3. Scenario 2 is basically the same: [lock(), act(), act(), unlock()] x 3.

	Scenario 1	Scenario 2
1.	out lock	out lock
2.	wait in lock_true	wait in lock_true
3.	out act	out act
4.	wait in act_done	wait in act_done
5.	out unlock	out act
6.	wait in unlock_done	wait in act_done
7.	out lock	out unlock
8.	wait in lock_true	wait in unlock_done
9.	out act	out lock
10.	wait in act_done	wait in lock_true
11.	out unlock	out act
12.	wait in unlock_done	wait in act_done
13.	out lock	out act
14.	wait in lock_true	wait in act_done
15.	out act	out unlock
16.	wait in act_done	wait in unlock_done
17.	out unlock	out lock
18.	wait in unlock_done	wait in lock_true
19.		out act
20.		wait in act_done
21.		out act
22.		wait in act_done
23.		out unlock
24.		wait in unlock_done

7.3 Observations

7.3.1 Without the covert channel

If we don't use the covert channel to check the actions at the server side, the genetic algorithm often creates a classifier which doesn't even communicate with the server. In such a situation the classifier would immediately respond *lock_true* whenever the client requests a lock.

7.3.2 With covert channel, only scenario 1.

If we use the covert channel to measure the fitness more accurately, we see that a perfect solution is created within approximately 11 generations.

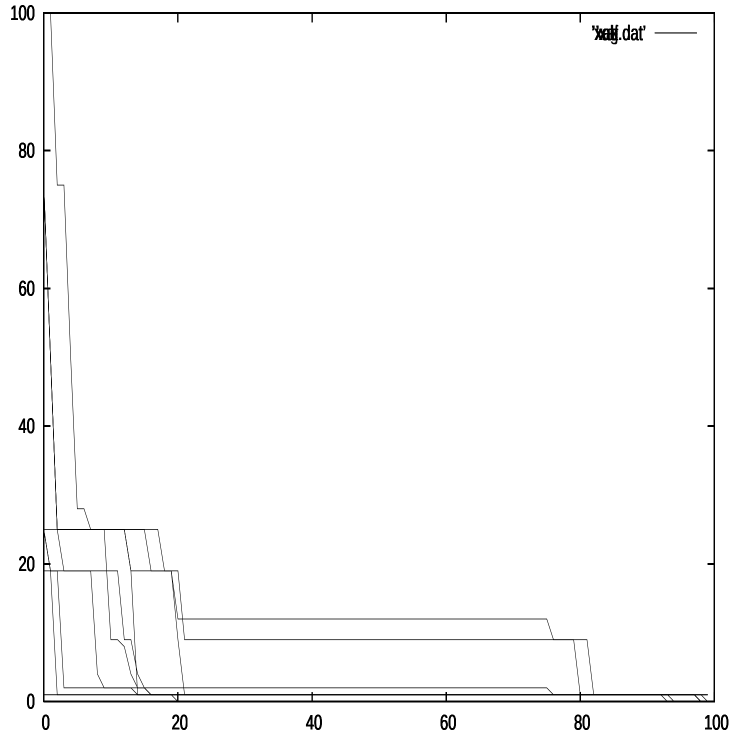


Figure: Fitness of the individuals in each generation.

7.3.3 Anticipation of what will happen

One of the learned classifiers developed a strange behaviour. Its fitness was 100%, but it worked asynchronously, as we illustrate in the code below. At the moment an *act* request arrived from the client it sent this message to the server, which would respond with *act_ done*. In response to this *act_ done* the adaptor would automatically send back an *unlock* operation to the server. The server would respond with *unlock_ done*, to which the classifier would send an *act_ done* to the client. The client in its turn now send a *lock_ request* which immediately results in an *unlock_ done* from the classifier.

This example shows how such a learned algorithm can anticipate future behaviour.

```
client.act -> server.act
server.act_done -> server.unlock
server.unlock_done -> client.act_done
client.unlock -> client.unlock_done
```

7.3.4 Too much anticipation of what will happen

A problem we encountered was the fact that sometimes, the adaptor anticipates too much and after the first *act*, doesn't stop *acting* anymore. We solved this problem by assigning a fitness of zero to such solutions.

7.3.5 Fitness measured with scenario1 and scenario2

If we measure the fitness with both scenarios, we avoid a lot of the (wanted or unwanted) anticipated behaviour. After maximum 10 generations we find a perfect solution.

Conclusion & Future Work

In this paper we gave an overview of how we can create intelligent interfaces which can be used in open distributed systems. The case studied was concurrency because this is known to be a major problem in peer to peer computing. It is clear that this approach is not restricted to the domain of concurrency.

Ou approach lets a genetic algorithm learn a classifier system. This classifier system contains classifiers which react upon the context they receive from client and server. The context is defined as a combination of the possible client-side and server-side states.

Future work:

- Doing more rigorous experiments. Trying Turing complete classifier-systems and see how they evolve.
- Work with larger interfaces. The interfaces we used were rather simple.
- The test-sets form another problem. How should one create a suitable test-set which covers all the necessary cases ?

Acknowledgements

I would like to thank my promotor Theo D'Hondt for the discussions we had on this subject and for giving me the time to investigate this matter. Thanks to Johan Fabry for last minute proof reading (/rescue).

References

[BFDV01] Werner Van Belle, Johan Fabry, Theo D'Hondt, and Karsten Verelst. *Experiences in Mobile Computing: The CBorg Mobile Multi-Agent System.* IEEE Computer Society Press, March 2001.

[BU00] Werner Van Belle and David Urting. *The Component System.* Deliverable for the SEESCOA project, Oktober 2000.

[Koz92] John R. Koza. *Genetic Programming; on the programming of computers by means of natural selection.* The MIT Press, 1992.

[Lea00] Doug Lea. *Concurrent Programming in Java (2nd edition) Design Principles and Patterns.* The Java Series. Addison Wesley, 2000.

[Pol96] Riccardo Poli. *Introduction to Evolutionary Computation.* School of Computer Science; The University Of Birmingham, 11 October 1996.

[Rei] W. Reisig. *An Informal Introductiopn To Petri Nets.* Humboldt University of berlin.