An Experiment In Automatic Adaptor Generation by means of Genetic Algorithms

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

Theo D'Hondt
tjdhondt@vub.ac.be

25th September 2005

http://borg.rave.org/
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 agents which can interface with other agents because it is an excellent area for interface conflicts. Conflicting interfaces can be remedied by using adaptors. It is possible to automatically generate those glue-adaptors by means of genetically engineered classifier systems, which use Petri-nets as a model for the underlying interfaces. This paper reports on an experiment that illustrates this approach.

1 Introduction

Mobile multi agent systems [CDGM] 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 usually are written by different developers and different organisations. Over time, this results in interface conflicts, which makes it difficult to create applications which keeps working without a lot of reintegration.

In comparison to standard desktop software this is nothing new, only, when working with agents, the scale of the problem differs. In standard software development we deploy a certain application using a number of components and whenever new components come up, we reintegrate our application and release a new version. With software agents this is not

possible because 1) we don't necessarily know with which agents our agents will communicate when executing on the net and 2) agent implementations can change over time without prior notification, thereby altering the offered API in a syntactic or semantic way. Whenever this happens, agents using this 'altered' API, will not work anymore. This will typically 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, with their very dynamic nature, may just be too impractical for programmers to use.

Genetic algorithms can provides us with a mean to automatically create program code for those adaptors and solve a number of interface conflicts automatically.

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, next we explain what kind of (typical) interface conflict we use in our experiment. Then we explain what a typical genetic algorithm looks like, which leads us to section 5 where we discuss the used model. Section 6, 7 then shows how this works on our previously introduced case.

2 Mobile Multi Agent Systems

Regarding the terminology *mobile multi agent system* there is some confusion. A multi agent system in AI [CDGM] denotes a software system that simulates the behaviour 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 execute [BFDV01]. We adopt the second definition. 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.

A requirement for automatic adaptor generation is that we should be able to place at runtime adaptors between two agents in a completely transparent way. This can only be done if a number of requirements are met:

1. ***Implicit Addressing:*** No agent can use an explicit address of another agent. This means no IP-number, DNS-name or alike can be used by agents. So we need to work in a connection oriented way.

The connections are set up solely by one agent: the connection broker. The connection broker will also place adaptors where necessary. Jini[Edw99] can be seen as a good example of this approach.

2. ***Disciplined Communication:*** No agent can communicate with other agents by other means than the ports. An agent should not open a socket himself, or use files on disk to communicate with other agents. All communication should pass through the ports.

3. ***No Shared Memory:*** All messages passed over a connection should be copied. Messages cannot be shared by agents (even if they are on the same host), because this would result in concurrency problems.

So, our mobile multi agents can be compared to processes [Mil99]. 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. In our system, agents communicate asynchronously and always copy their messages completely upon sending. This, because synchronous communication can be easily expressed with asynchronous constructs, while the opposite is difficult.

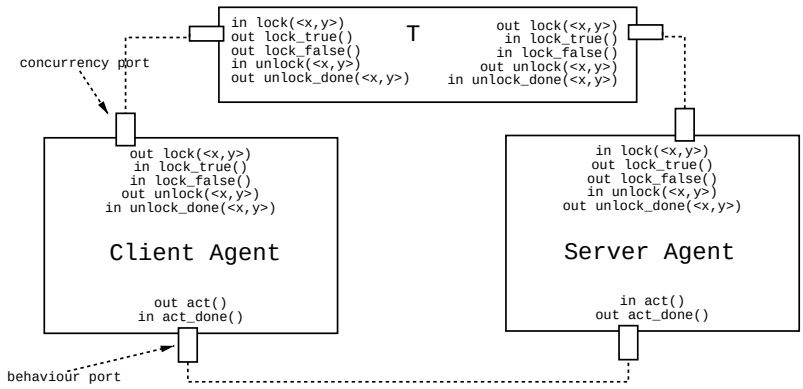

Figure 1: Client and server agent, both honouring the same interface, whereby the client agent *sends out* lock requests and the server agent can accept *incoming* lock requests. Agents interact with each other over ports. This enables us to easily place adaptors in between them. (e.g., Adaptor T)

The connections between agents are full duplex. This means that every agent can *send* and *receive* messages over a port. This brings us in a situation

where a certain interface is *required* from another agent and where an agent can *provide* a certain interface (see figure 1).

The above view on agents with the given requirements are no restraint. In fact, most mobile multi agent systems use these requirements because they are inherent distributed systems and typically distributed systems are bound by such a constraints.

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 provides a concurrency interface (typically a transaction interface) [Lea00] 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 provides a concurrency interface is often a problem. It can provide a full-fledged optimal transaction interface or it can provide a simple lock/unlock interface. How two interfaces of a different kind interface with each other gives rise to problems.

In our example we use a simple lock/unlock interface of the server 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 or 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 server agent also offers some behaviour on another port. This behaviour is as simple as possible.

```
port behaviour
incoming act(resource)
```

```
outgoing act_done(resource)
  // will do some action on the agent.
```

Even with this simple definition of a concurrency interface, the semantics can be implemented in different ways. We will use two kinds of concurrency implementations:

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

Binary Semaphores 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 considers *lock* and *unlock* can give rise to interface conflicts. In figure 2 the client agent expects a counting semaphore from the server agent. The server agent offers binary semaphore. The client can lock a resource twice and expects that the resource can be unlocked twice. In practice the server just has marked the resource as *locked.* If the client now unlocks the resource, the resource will be unlocked. Acting upon the server now is impossible, while the client expects it to be possible.

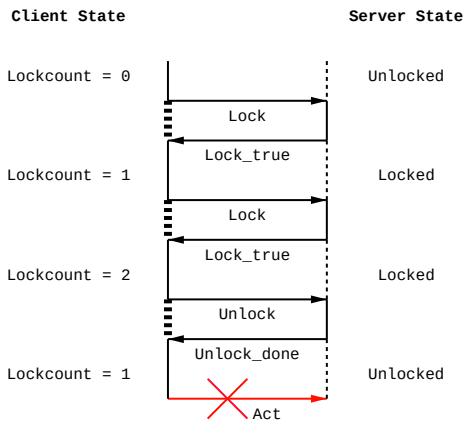

Figure 2: An interface conflict when the client agent expects a counting semaphore from the server agent and the server agent only offers a binary semaphore.

In the remaining of the paper we will try to generate an adaptor agent between both locking strategies automatically. This will be achieved by means of genetic algorithms, which are introduced in the next section.

4 Genetic Algorithms

A genetic algorithm[Koz92] is an algorithm which tries to solve a problem by trying out a number of possible solutions. Every solution is measured with respect to the problem at hand. From the best solutions, new solutions are created. This process is repeated until a suitable solution is found. The solutions to a problem are called *chromosomes* or *individuals*. Every individual has a number of parameters which can be modified. This are the *genes*. A *generation* is a set of individuals which all try to solve the same problem(s). How well an individual performs is called the *fitness* of the individual.

When a generation is measured we keep (*reproduce*) 10% of the best individuals. We throw away 10% of the worst individuals (not fit enough) and add *cross-overs* from the 10% best group. A cross-over is a random selection of genes from each individual. The other 80% are simply *mutated*, which means that the genes of each individual are simply changed at random.

The standard question before implementing any genetic algorithm are

1. What are the individuals ? What are the genes of the individuals (ie, the parameters that can change for every individual) ?
2. How do we represent the individuals ?
3. How do we measure the fitness of an individual ?
4. How do we initially create individuals ? How do we mutate them and how do we create a cross over of two individuals ?

In our case the individuals are the adaptor agents that we will place between communicating agents. Representing the programs we place in such an adaptor agent can be done in a number of different ways: we can use Scheme, Java or other human-used languages as a representation. The problem with these is that they are very verbose and the valid program space is very small. Therefore we choose another (well known) format for our programs: classifier systems. These will be explained in section 6.

A problem of genetic algorithms, and learning systems in general, is that they cannot learn what they do not see. How can we make sure that our adaptors/individuals have enough input from the environment to make correct decisions ? This will be discussed in the next section.

5 Petri-nets

How can we make sure that our concurrency adaptors have enough input from the environment ? Our concurrency adaptors should at *least* know

1. which state the client agent *expects* the server to be in. (or the *projected* client-state).

2. in which state the server agent is.

If we don't know that the client thinks that it still has a lock, and we don't know the state of the server: unlocked (see figure 2), no learned algorithm can make a correct decision.

Representing the state and state-transitions of an agent will be done by means of Petri-nets [Rei00]. 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.

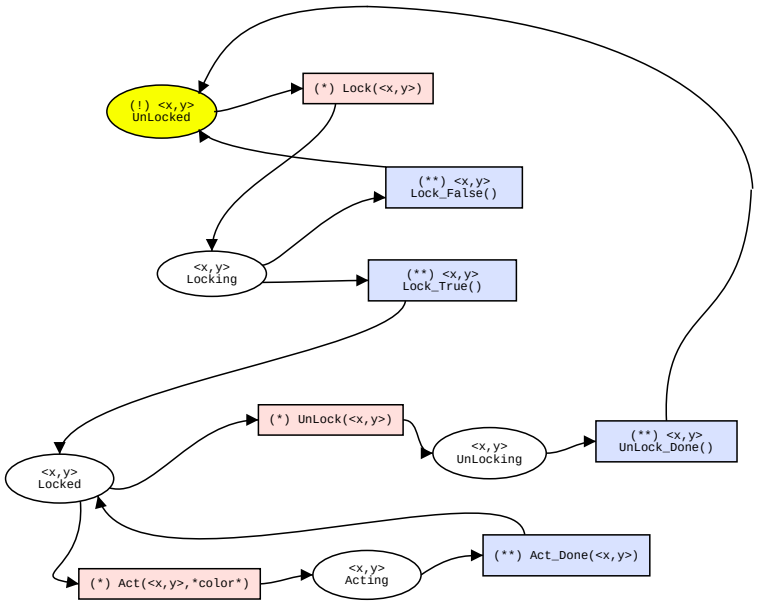

Figure 3: Binary semaphore locking strategy, described in a Petri-net. The red boxes (marked with '*') are the messages which either comes in are goes out. The blue ones (marked with '**') are exactly the opposite.

For example, in figure 3 one sees how a binary semaphore locking strategy is modelled. States are shown as circles, while transitions are shown as boxes. The current state *unlocked* is coloured in yellow (marked with a '!'). From this state the agent *requiring* this interface, can choose only one action: *lock*. When it chooses this action, it is in the *locking* state until *lock_ true* or *lock_ false* comes back. Note that we can also use this Petri-net to model the behaviour of an agent which *provides* this 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* is sent back to the client when making the transition.

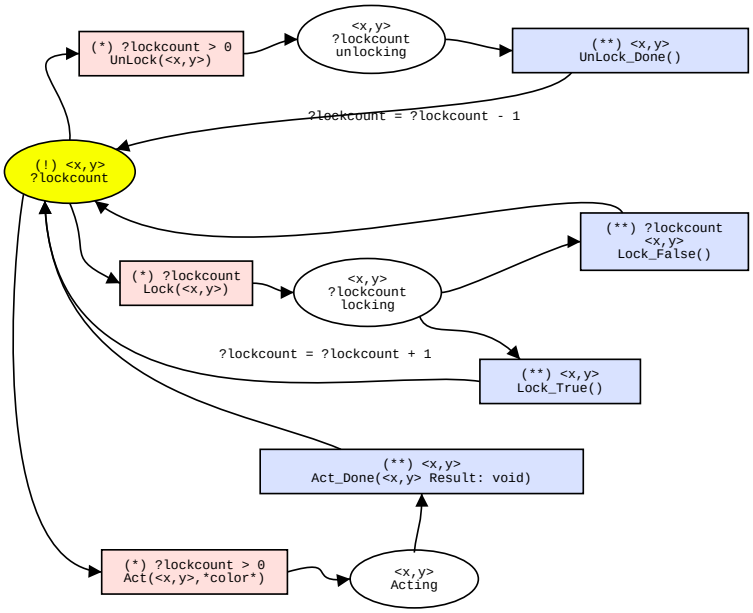

Figure 4: Counting semaphore locking Strategy described as a Petri-net.

Figure 4 represents the counting semaphore locking strategy. One sees how a transition can only be made when a certain condition holds true: an agent can only *unlock* when it has a lock count which is larger than 0. The agent state holds this extra field: ?*lockcount*.

In the next section we will discuss the representation we will use for our adaptors (the individuals in the genetic algorithm). To be able to make correct decisions this representation needs enough input from its environment. In our experiment we use the petri-net states as input for our adaptors.

6 Classifier Systems

Adaptors should ease the semantic differences between two agents. Since we want to learn this we need to specify our programs in some kind of way. We choose to use classifier systems for this because they are simple, contains a memory and are Turing complete.

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 classifier list takes a message list as input and produces a new message list as output. Every message in the message list is a set of bits (0 or 1) which are matched against a set of classifiers. A classifier rule (classifier for short), contains a number of (possible negated) conditions and an action. We will work with four conditions.

Conditions and actions are both ternary 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 (see figure 5).

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

Figure 5: Illustration of how actions produce a result when the condition matches. ~ is negation of the next condition. In this illustrative example only two conditions are used.

Such a classifier system will need to reason about the actions to be performed based on the available input. The input of a classifier system consists of a full description of the client state, a full description of the server state and possible the requested actions from either the client agent or the server agent. Every message needs a suitable representation

We model the agent states as classifier messages. The representation will not contain the Petri net states, but instead, the Petri net transitions that are possible, because in essence they represent the same. If we specify what a Petri net can do in a certain state, then we also now in which state it is. Hence, as for the information they are the same. The reason why we choose this representation lies in the possible optimalisations: we immediatelly know

what kind of actions are possible; something we would have to learn ourselve if we would only use the actual state.

The agent actions are represented in the same way.

Our classifier system requires 4 conditions. The first condition must be an incoming action to match, the second condition must match a client state representation, the third condition must match a server state representation and the last condition is a freeform condition, which can be used by the classifiersystem to check its own working memory.

With these semantics for the messages, simply translating a request from the client to the server requires 1 rule. Another rule is need to translate requests from the server to the client (see figure 6).

Condition	A1	Rule description
100 ### ...	111 ### ...	Every incoming action from the client (100) is translated into an outgoing action on the server(111)
101 ### ...	110 ### ...	Every incoming action from the server (101) is translated into an outgoing action to the client (110)

Figure 6: Blind translation between client and server agents.

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 there is a lock count that is larger than zero. Figure 7 shows how we can represent such a behaviour.

Condition1	Condition2	Action	Rule description
100 001 000 000	001 000 000 00**1**	110 010 000 000	If the client wants to lock and has a lock we send back a lock true
100 001 000 000	001 000 000 00**0**	### ### ### ###	If the client wants to lock and has no lock we immediatelly send the message through.

Figure 7: Translating a client agent lock request to a server agent lock action when necessary.

7 Adaptor Generation

As seen in section **4** a genetic algorithm requires individuals to test. We defined our individuals as classifier systems. They form the chromosomes of

the genetic algorithm while the classifier conditions and actions are the genes. This method of using full classifier systems is also known as the Pittsburgh approach [Smi80].

Individuals are initially empty. Every time a certain situation arises and there is no matching gene we will add a new gene. This gene, which is a classifier rule has a matching 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 individuals is done by randomly adapting a number of genes. A gene is adapted by selecting a new random action.

To create a *cross-over* of individuals we iterate over both classifier lists and each time randomly select a rule that will be stored in the resulting chromosome.

Fitness is measured 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 an individual is determined by how far the scenario can execute without unexpected behaviour. Of course this is not enough, we should not have solutions that 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 needs a covert channel over which it can contact the server to verify its actions.

8 The Experiment

8.1 Setup

The experiment is set up as a connection broker between two agents. The first agent 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 produce a test-client and a test-scenario. The server will produce a test-server. In comparison with the original agent, these testing agents have 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; i.e., a correct adaptor has been found, the connection is set up.

8.2 Scenarios

The scenarios offered by the client are the ones that will determine what kind of classifier system is generated. We have tried this with three scenarios. Scenario 1 is a sequence: [`lock(), act(), unlock()`] x 3. (See figure 8) Scenario 2 is basically the same: [`lock(), act(), act(), unlock()`] x 3. The reason why we added such a second look-alike scenario will become clear in our observations. The third scenario is the case we explained earlier in the paper (see figure 2).

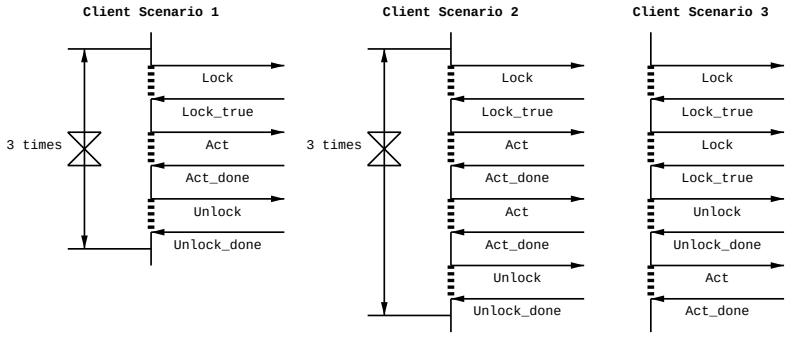

Figure 8: The three test scenarios we used to measure the fitness of an adaptor. The dashed vertical lines are waits for a specific message (e.g., Lock_true).

8.3 Observations

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.

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. In this case the fitness of each individual was measured by scenario 1 and scenario 3. See figure 9.

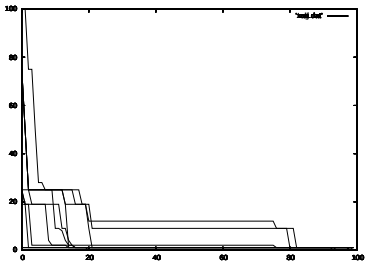

Figure 9: Fitness of the individuals in each generation. Horizontally we find 100 individuals. Vertically we find the fitness of the individuals (after sorting). Every line represents a generation: the higher the line, the younger the generation.

Anticipation of what will happen One of the learned classifiers developed a strange behaviour. Its fitness was 100%, but it worked asynchronously, as illustrated in figure 10. 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.

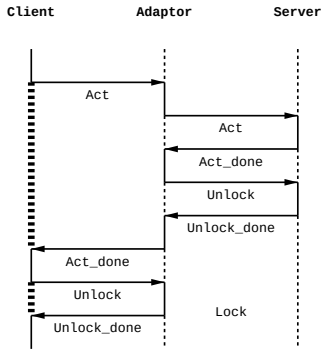

Figure 10: How an adaptor anticipates the next request from the client and afterwards returns the already available correct response.

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

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*, keeps on *acting*. We solved this problem by assigning a fitness of zero to such solutions.

Fitness measured with all scenarios If we measure the fitness with all three scenarios, we avoid a lot of the unwanted anticipated behaviour, while retaining the wanted anticipated behaviour. After maximum 10 generations we found a perfect solution.

From this experiment we can see the strengths of this approach:

- We specify programs by means of a test set. This resembles the behaviour of a standard software developer. At the moment a certain unwanted behaviour (a bug) pops up, het starts debugging. With this approach the debugging is done automatically.
- Automatically generated adaptors can be better than hand crafted adapters since they can anticipate behaviour.

9 Conclusion & Future Work

In this paper we reported on an experiment to show how we can create intelligent interfaces to make mobile multi agents communicate.

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

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.

As for future work there remain some important problems. First of all we need to do more rigorous experiments. This includes, using larger interfaces (with more functions, hence larger petri-nets).

A second problem involves the use of the test scenarios. The test scenarios should cover all the actions which will be invoked upon the server in all possible combinations. How can we write good tests which doesn't leave any open holes for the programmer. And if we can, are the Petri-nets still necessary ? In other words: can we learn the model automatically just by looking at the interaction between the agents.

A third (all-round) problem is the representational mapping. In our example we use two alike representations: Petri nets represented by a bit pattern. This makes learning an adaptor easy. To make this algorithm stronger we need a representational mapping which is able to translate one state to another state.

Acknowledgements

Thanks to Johan Fabry, Tom Tourwé and Tom Mens for proofreading this paper.

References

[BFDV01] W. Van Belle, J. Fabry, T. D'Hondt, and K. Verelst. Experiences in mobile computing: The CBorg mobile multi-agent system. In Wolfgang Pree, editor, *Proc. TOOLSEE 2001.* IEEE Computer Society Press, March 2001.

[BU00] W. Van Belle and D. Urting. *The Component System.* Technical Report 3.4 for the SEESCOA project, October 2000.

[CDGM] A. Chavez, D. Dreilinger, R. Guttman, and P. Maes. *A Real-Life Experiment in Creating an Agent Marketplace.* Proc. Int'l Conf. Practical Application of Intelligent Agents and Multi-Agent Technology, London, UK, April 1997.

[Edw99] W. K. Edwards. *Core Jini.* The Sun Microsystems Press Java Series. Prentice Hall, 1999.

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

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

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, May 1999.

[Pol96] R. Poli. Introduction to evolutionary computation. Technical report, School of Computer Science; University Of Birmingham, October 1996.

[Rei00] W. Reisig. *An Informal Introduction To Petri Nets.* Proc. Int'l Conf. Application and Theory of Petri Nets, Aarhus, Denmark, June 2000.

[Smi80] S.F. Smith. *A Learning System Based on Genetic Adaptive Algorithms.* PhD thesis, Department of Computer Science, University of Pittsburgh, 1980.