Automatic Adaptor Generation for Mobile Agents by means of Genetic Algorithms

Not for distribution or attribution: for review purposes only.

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

Tom Mens†
tom.mens@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

30th August 2002

Vrije Universiteit Brussel - Programming Technology Lab
Pleinlaan 2, 1050 Brussel - Belgium
Tel: +32 486 68 84 48 Fax: +32 2 629 3525

Abstract

Mobile multi-agent systems can be seen as a basis for global peer-to-peer computing. This new computation paradigm becomes increasingly more important as mobile devices become more powerful. Unfortunately, an open internet environment is an excellent area for interface conflicts between agents that want to communicate with each other. Although conflicting interfaces can be remedied by using adaptors, the number of possible combinations of different interfaces increases dramatically. Therefore we propose a technique to generate interface adaptors automatically. This is achieved by means of genetically engineered classifier systems (a well-known genetic algorithm technique) that use Petri nets as a specification for the underlying interfaces. This paper reports on an experiment that validates this approach. For designers of mobile applications, our approach is an important step forward, since the task of the developer is shifted from the writing of adaptors to the specification of test scenarios.

Key terms. mobile agents, interface conflicts, peer-to-peer computing, genetic algorithms, classifier systems

1 Introduction

In the world of distributed computing, a *mobile multi-agent infrastructure* is an environment in which multi agents can execute [BFDV01]. Mobile multi agents are active autonomous software components that communicate with other agents to reach specific goals. They can migrate to other agent systems, thereby carrying their program code and data along. Mobile multi-agent systems are a good basis for global peer-to-peer computing [CDL$^+$96]. Since

*Corresponding author. He is developing peer-to-peer embedded systems for a project funded by the Flemish Institute for Science and Technology (IWT).

†Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)

mobile agents are usually provided by different developers and different organisations, communication interfaces provided (and required) by those agents can vary greatly. As a result interface conflicts arise.

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 (they are mobile) and 2) agents evolve: their implementations can change over time without prior notification, thereby altering the offered API (Applications Programmers Interface) or its behaviour. Whenever this happens, agents communicating with this syntactically or semantically modified agent will cease to work, typically resulting in a cascaded application breakdown.

On first sight, a solution to this problem would be to offer interface adaptors between every possible pair of agents. The problem with this approach is that the number of adaptors grows quadratic to the number of agent interfaces and as such it simply doesn't scale. The solution is to automate the generation of interface adaptors between communicating agents.

As potentially useful techniques for this adaptor generation, we explored the research domain of adaptive systems. We found that the combination of genetic algorithms, classifier systems, and a formal specification in terms of Petri nets allowed the automatic detection of interface conflicts, as well as the automatic creation of program code for interface adaptors that solve these conflicts. This paper reports on an experiment we performed to validate this claim.

2 Prerequisites of Mobile Multi-Agent Systems

In the mobile agent system we propose, our agents can be compared to processes [Mil99]. Agents communicate with each other only by sending messages over a communication channel.Communication channels are accessed by the agent's ports. Agents communicate asynchronously and always copy their messages completely upon sending. The connections between agents are full duplex: every agent can *send* and *receive* messages over a port. This brings us in a situation where an agent *provides* a certain interface and *requires* an interface from another agent. An agent can have multiple communication channels: one for every communication partner, or more than one for every provided/required interface.

The above view on agents with the given requirements is no restraint. In fact, most mobile multi-agent systems (e.g., Aglets [Som97], Mole [BHRS98]) use these requirements because they are inherently distributed systems, which are typically bound by such constraints. Other requirements that are imposed on an agent system to allow us to generate interface adaptors are enumerated below:

1. ***Implicit Addressing:*** No agent can use an explicit address of another agent. Agents work in a connection-oriented way. The connections are set up solely by one agent: the connection broker. This connection broker will also place adaptors where necessary.

2. ***Disciplined Communication:*** No agent can communicate with other agents by other means than its ports. Otherwise, 'hidden' communication (e.g., over sockets) cannot be modified and the adaptor has not all the access it needs.

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 a massive amount of concurrency problems.

4. ***Explicit Interfaces Descriptions***: For humans, a semantic interface description would be some form of documentation written in natural language. For computers we need a more formal, explicit, description of the semantics of an interface. A simple syntactic description is no longer suitable.

3 Specifying Interfaces

As a running example throughout this paper we choose a typical problem of open distributed systems: how agents synchronise with each other. In closed distributed systems this is less of a problem. A server provides a concurrency interface (typically a transaction interface) [Lea00] that can be used by clients. The clients have to adhere to this specification or they won't function. Since the server concurrency API doesn't change that much this is a workable 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. For example, it can provide a full-fledged optimal transaction interface or it can provide a simple lock/unlock interface. When two such interfaces of a different kind interact, we can run into problems of incompatible interfaces.

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 descriped for the server as follows. Please note that a similar description can be given for the clients.

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

The semantics of this interface can be implemented in different ways. We will use two kinds of locking semantics [Lea00]:

Counting semaphores allow a client to 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 the counter reaches zero. These semantics allow us to use routines which autonomously lock resources.

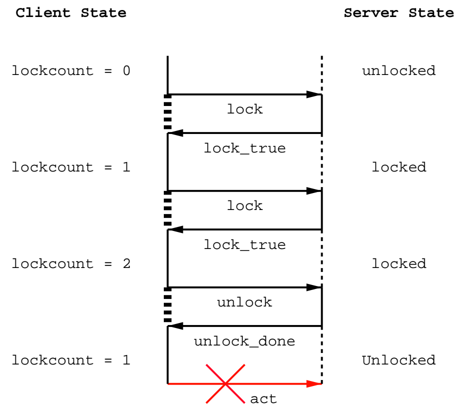

Figure 1: An interface conflict when the client agent expects a counting semaphore from the server agent and the server agent only offers a binary semaphore.

Binary semaphores provide a locking semantics that doesn't offer a counter. It simply remembers who has locked a resource and doesn't allow a second lock. When unlocked, the resource becomes available again.

Differences in how the API considers *lock* and *unlock* can give rise to interface conflicts. In figure 1 the client agent expects a counting semaphore from the server agent, but the server agent offers a 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.

The above interface conflict arises because the API does not specify enough semantic information. Hence, we propose to use a more detailed and generally applicable formalism, namely Petri nets, to offer an explicit description of the interface semantics. Petri nets [Rei00] offer a model in which we can have multiple agents that each go from state to state by means of state transitions. In the context of our locking example, this allows us to write a suitable interface adaptor by relying on:

1. which state the client agent *expects* the server to be in
2. in which state the server agent is.

Both kinds of information are essential: If we don't know that the client thinks that it still has a lock, and we don't know that the state of the server is unlocked (see figure 1), no learned algorithm can make a correct decision.

As an example Petri net that offers the needed additional semantics, the left part of Figure 2 specifies a binary semaphore locking strategy. The current state is *unlocked*. From this state the agent *requiring* this interface, can choose only one action: *lock*. It then goes to 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 that *provides* this interface. It is perfectly possible to offer an interface that 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.

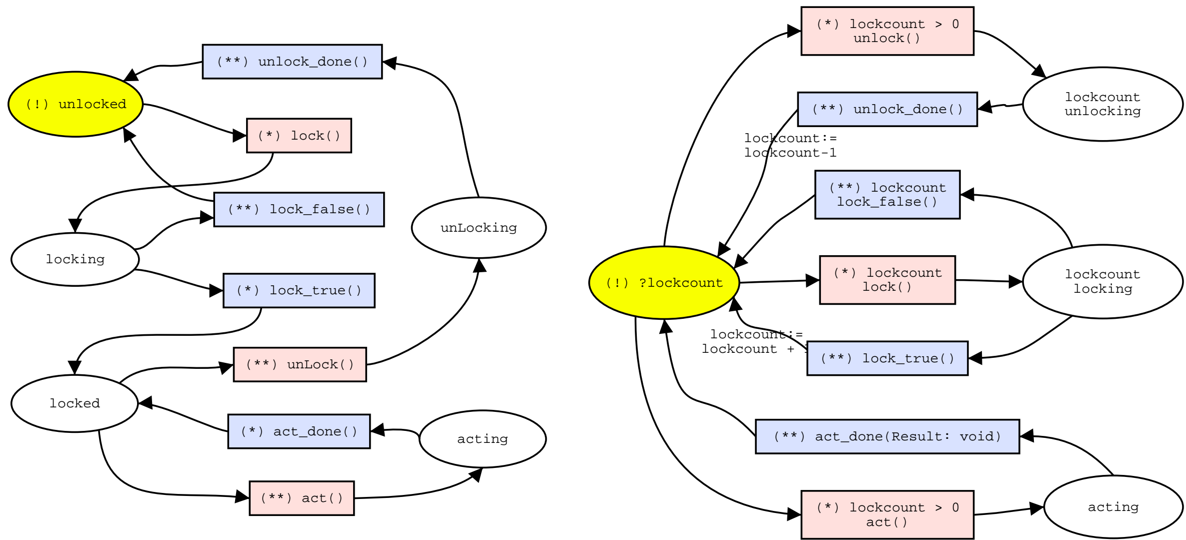

Figure 2: Two Petri-net descriptions of agent interfaces. Ellipses correspond to states. Rectangles correspond to transitions. The current state (marked with '!') is coloured yellow. The red transitions (marked with '*') represent incoming messages. The blue ones (marked with '**') represent outgoing messages.

The Petri net on the right of figure 2 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 strictly positive lock count. The agent states store the value of this *lockcount* variable. After unlocking, the value of this variable is decreased by one.

4 Learning Interfaces

Interface adaptors overcome the semantic differences between two agents. Given the virtually infinite number of agent interactions that are possible, it is not feasible to define an interface adaptor manually for every possible combination of interacting agents. Hence, we need a way to generate or learn these interface adaptors automatically.

4.1 Genetic Algorithms

We propose to use genetic algorithms with classifier systems for the purpose of learning interface adaptors. Classifier systems are known to work very well in cooperation with genetic algorithms, they are Turing complete and they work very well with *symbols*. This is important because our Petri net description is in essence a symbolic representation of the different states in which an agent can reside. If this represenation would be *numerically,* techniques such as neural networks[KvdS96], reinforcement learning and Q-learning [SB98] could probably be used. Work that also uses the classifier systems approach, where a robot learns to interact with a user by means of an evolved classifier system is [SDD99].

A *genetic algorithm* [Glo89] is an algorithm that tries to solve a problem by trying out a number of possible solutions, called *individuals*. Every individual is an encoding of a number of modifiable parameters called *genes*. Every individual is assigned a *fitness* that measures how well the individual solves the problem at hand. From the solutions with the best fitness,

a new *generation* of solutions is created. This process is repeated until the most suitable solution is found.

The standard questions before implementing any genetic algorithm are: What are the individuals and their genes? How do we represent the individuals? How do we define and measure the fitness of an individual? How do we initially create individuals? How do we mutate them and how do we create a cross over of two individuals? How do we compute a new generation from an existing one?

In the next subsection, we will provide an answer to these questions in our particular implementation.

4.2 Our implementation

In our implementation, the individuals will be interface adaptors between communicating agents. The question of how to represent these individuals is more difficult. We could use well-known programming languages (such as Scheme or Java) to represent the behaviour of the adaptor. Unfortunately, the inevitable syntactic structure imposed by these languages complicate the random generation of programs. Moreover, these programming languages do not offer a standard way to access memory. An alternative that is more suitable for our purposes is the Turing complete formalism of *classifier systems* [**?**]. This method of using full classifier systems as individuals is known as the Pittsburgh approach [Smi80][1] A classifier system is a kind of control system that has an input interface, a finite message list, 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 fixed-length binary string that is matched against a set of classifier rules. A classifier rule contains a number of (possible negated) conditions and an action. These conditions and actions form the genes of each individual in our genetic algorithm. Conditions and actions are both ternary strings (of 0, 1 and #). '#' is a pass-through character that, in a condition, means 'either 0 or 1 matches'. If found in an action, we simply replace it with the character from the original message. Table 1 shows a very simple example. When evaluating a classifier system, all rules are checked (eventually parallel) with *all* available input messages. The result of every classifier evaluation is added to the end result. This result is the output message list. For more details, we refer to [Glo89].

A classifier system needs to reason about the actions to be performed based on the available input. In our implementation, the input of a classifier system consists of a ternary string representation of the client state and server state (as specified by the Petri net), as well as a ternary string representing the requested Petri net transition from either the client agent or the server agent.

With these semantics for the classifier rules, translating a request from the client to the server requires only one rule. Another rule is needed to translate requests from the server to the client (see table 2).

The number of bits needed to represent the transitions depends on the number of possible state transitions in the two Petri nets of figure 2. The number of bits needed to represent the states of each Petri net depends on the number of states in the Petri net as well as the

[1]Another well-known approach is the Michigan approach, whereby a set of classifiers evolves together to reach a solution. This approach is not suitable for our purposes because one rule doesn't cover the behaviour of an adaptor, as such cross-over between single rules would not help that much [BF93].

Input message list = { 001, 101, 110, 100 }

Condition		Action	Matches	Result
00#	101	111	yes	111
01#	1##	000	no	/
1##	~00#	###	no	/
1##	###	1#0	yes	100, 110

Output message list = { 111, 100, 110 }

Table 1: Illustration of how actions produce a result when the conditions match all messages in the input message list. ~ is negation of the next condition. In this illustrative example, a disjunction of two conditions is used for each classifier rule. The second rule does not match for input message 001. The third rule does not match because the negated condition is not satisfied for input message 001.

classifier condition			action	rule description
requested transition	client state	server state	performed action	
00####	####	###	11#...#	Every incoming action from the client (00) is translated into an outgoing action on the server (11)
01####	####	###	10#...#	Every incoming action from the server (01) is translated into an outgoing action to the client (10)

Table 2: Blind translation between client and server agents. The last 5 characters in column 1 represent the corresponding transition in the Petri net. The characters in the second and third column represent the states of the client and server Petri net, respectively. The fourth column specifies the action to be performed based on the information in the first four columns.

variables that are stored in the Petri net (e.g., the *lockcount* variable requires 2 bits if we want to store 4 levels of locking).

Although this is a simple example, more difficult actions can be represented. Consider the situation where the client uses a counting-semaphores locking strategy and the server uses a binary-semaphores locking strategy. In such a situation we don't want to send out the lock-request to the server if the lock count is larger than zero. Table 3 shows how we can represent such a behaviour.

The genetic algorithm we implemented uses a full classifier list with variable length. The classifier list is an encoding of the Petri nets, as representation for the *individuals*. Every individual is initially empty. Every time an individual encounters a situation where there is no matching gene a new gene (i.e., a new classifier rule) will be added with a condition that covers this situation and a random action that is performed on the server and/or the client. This way of working, together with the use of Petri-nets guarantees that the genetic algorithm will only search within the subspace of possible solutions. Without these boundaries the genetic algorithm would take much longer to find a solution.

classifier condition			action	rule description
requested transition	client state	server state	performed action	
00 001	~##00	###	10 010 ...	If the client wants to lock (001) and already has a lock (~###00) we send back a lock_true (010)
00 001	##00	###	11 001 ...	If the client wants to lock (001) and has no lock (###00) we immediately send the message through (001).

Table 3: Translating a client agent lock request to a server agent lock action when necessary.

parameter	value
learning-time	offline
genotype	classifier system represented as bitstring
population size	100
maximum generations	11
parent selection	ranking selection (10 % best)
mutation	bitflip on non ranked individuals
mutation rate	0.8
crossover	uniform
crossover rate	0.1
input/output interfacing	petri net state/transition representation
actions	message sending

Table 4: Parameters and charasteristics of the genetic algorithm

Fitness of an individual is measured by means of a number of test scenarios (which will be discussed in the next section). Every test scenario illustrates a typical behaviour the client requests from the server. The fitness of an individual is determined by how many actions the scenario can execute without yielding unexpected behaviour. Of course this is not enough; we should not have solutions that completely shortcut the server. For example, the algorithm could return *lock_ true* every time a request comes in from the client, without even contacting the server. To avoid this kind of behaviour our algorithm provides a covert channel that is used by the test scenario to contact the server to verify its actions.

The genetic algorithm is a steady state GA, with a ranking selection criteria: to compute a new generation of individuals, we keep (*reproduce*) 10% of the individuals with the best fitness. We throw away 10% of the worst individuals (not fit enough) and add *cross-overs* from the 10% best group[2]. 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 individual. It should be noted that the individuals that take part in cross over are never mutated. The remaining 80% of individuals are *mutated*, which means that the genes of each individual are changed at random. With random we mean that for every rule a new action to be performed on server or client is chosen and on top of this, in 50% of the classifiers, one bit of the state representations is generalised by replacing it with a #, this allows the genetic algorithm to find solution for problems that are not presented yet.

5 The Experiment

5.1 Setup

We will now present the experiment that shows that it is possible to use the above techniques to automatically learn an interface adaptor between incompatible locking strategies.

The experiment is set up as a connection broker between two agents. The first agent contacts the second 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 so by requesting a test agent from both parties. The client will produce a test client and test

[2] These values were taken from [Koz92] and gave good results during our experiments.

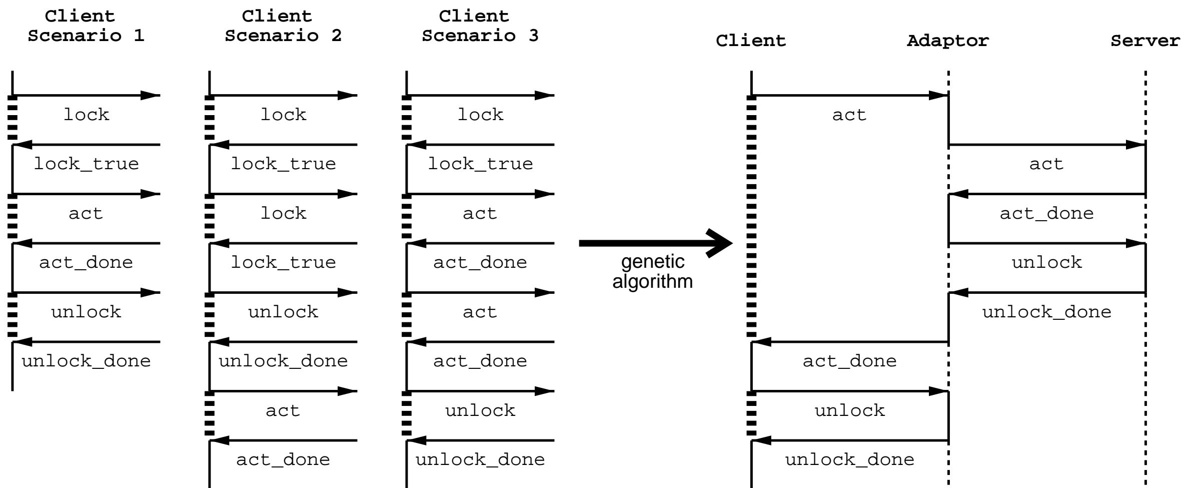

Figure 3: The three test scenarios we used as input to the genetic algorithm. The dashed vertical lines are waits for a specific message (e.g., lock_true). The resulting adaptor in this case anticipates the next request from the client and afterwards returns the already available correct response.

scenarios. 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 set of 100 individuals (i.e., adaptor agents), will deploy 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.

The scenarios offered by the client are the ones that determine what kind of classifier system is generated. We have tried this with three scenarios, as illustrated on the left of figure 3. Scenario 1 is a sequence: [`lock(), act(), unlock()`]. Scenario 2 is the case we explained in figure 1. Scenario 3 is similar to scenario 1: [`lock(), act(), act(), unlock()`]. The reason why we added such a look-alike scenario will become clear in our observations. In all three scenarios, we issue the same list of messages three times to ensure that the resource is unlocked after the last unlock operation.

5.2 Observations

By examining the results of applying our genetic algorithm we can make the following observations:

- If we use the covert channel to measure the fitness, we see that a perfect solution is created within maximum 11 generations. (After executing the genetic algorithm 100 times). If we don't use the covert channel to check the actions at the server side, the genetic algorithm often creates a classifier that doesn't even communicate with the server. In such a situation the classifier immediately responds *lock_true* whenever the client requests a lock.

requested transition	client state	server state	performed action	description
00 00 01	10 00 #1	11 ####	00 10 01	client?lock() & client=locked-> client.lock_true()
00 00 01	10 00 1#	11 ####	00 10 01	client?lock() & client=locked-> client.lock_true()
00 00 01	10 00 00	~ 11 010	00 11 01	client?lock() & client=!locked -> server.lock()
00 00 10	10 00 1#	11 ####	00 10 11	client?unlock() & clientlock>2 -> client.unlock_done()
00 00 10	10 00 01	11 ####	00 11 10	client?unlock() & clientlock=1 -> server.unlock()
00 00 11	10 #####	11 ####	00 11 11	client?act() -> server.act()
00 01 10	10 #####	11 ####	00 10 10	server?act_done() -> client.act_done()
00 01 00	10 #####	11 ####	00 10 00	server?lock_false() -> client.lock_false()
00 01 01	10 #####	11 ####	00 10 01	server?lock_true() -> client.lock_true()
00 01 11	10 #####	11 ####	00 10 11	server?unlock_done() -> client.unlock_done()
00 00 01	10 00 00	11 010	00 10 00	client?lock() & server=locked & client=!locked -> client.lock_false()

Table 5: The generated classifier system.

- One of the learned classifiers developed a strange behaviour. Its fitness was 100%, but it worked asynchronously. In other words, the adaptor would contact the server agent before the client agent even requested an action from the server agent. It could do so because the adaptor knew that the client would request that particular action in the given context. This is illustrated on the right of Figure 3. This implies that a learned algorithm can anticipate certain kinds of future behaviour.

- Initially, we only used scenario 1 and 2 to measure the fitness of each individual. A problem we encountered was the fact that sometimes, the adaptor anticipates too much and after the first *act*, keeps on acting. This problem was solved by assigning a zero fitness to such solutions.

- As a last experiment we measured the fitness by combining information from all three scenarios. This allowed the genetic algorithm to find a perfect solution with less generations because separate individuals that developed behaviour for a specific test scenario were combined in a later generation using cross-over. This illustrates the necessity for a cross-over operator. A random search would take considerably more time to find a combination of both behaviours.

The classifier system that is generated by the genetic algorithm when providing as input all three test scenarios is given in table 5. The produced classifier system simply translates calls from client to server and vice versa, unless it is about a lock call that should not be made since the server is already locked. The bit patterns in the example differ slightly from the bit patterns explained earlier. This is because we need the ability to make a distinction between a 'transition-message' and a 'state-message'. All transition messages start with 00 and all state-messages start with 10 for client-states and 11 for server-states.

5.3 Discussion

In our approach, the problem of 'writing correct interface adaptors' is shifted to the problem of 'specifying correct test sets': whenever the developer of a mobile agent encounters an undesired behaviour, he needs to specify a new test scenario that avoids this behaviour. This

test scenario is given as additional input to the genetic algorithm, so that the algorithm can find a solution that does not exhibit this behaviour. The result of this approach is that the programmer does not have to implement the interface adaptors directly, but instead has the responsibility of writing good test sets (i.e, a consistent set of scenarios that is as complete as possible). This is a nontrivial problem, since the test set needs to cover all *explicit* as well as *implicit* requirements. The main advantage of test sets over explicit interface adaptors is that we need a new interface adaptor for every pairs of interacting agents, while we only need one test set for each individual agent. As such, test sets are more robust to changes in the environment: when an agent needs to interact with a new agent, there is a good chance that the test set will already take into account potential interface conflicts. Another important advantage of test sets is that they can help in automatic program verification. Bugs in the formal specification (the Petri net) can be detected and verified at runtime. As such, this approach helps the developer to stay conform to the program specification. This clearly helps him in his goal to write better software.

Below we discuss some strengths and weaknesses of our approach:

- Since the interface adaptors need to be generated at runtime, efficiency is an important criterion. The current Java implementation takes about 20 seconds to check 10 generations of 100 individuals, after which a solution was always found. The algorithm can be optimised further using the inherent parallelism of open distributed system. Once the adaptor is generated the learning algorithm is no longer needed. As such, there is no performance penalty of any kind at runtime.

- Automatically generated adaptors can be better than hand-crafted adapters since they can reorder incoming and outgoing messages as necessary. This can result in anticipated behaviour that boosts performance.

- The genetic algorithm we present here has the problem that it needs to learn a certain behaviour based on a very small set of examples. The learning algorithm will automatically generate more general or more specific adaptors when offered test sets. This tendency to generalize matches does not always correspond to how a programmer tries to generalise. If there is a close correspondence, the programmer simply needs to write test sets that will naturally be generalised to the desired adaptor. On the other hand, if the generalisation (or specialisation) does not fit the developer's way of thinking, the algorithm will generate wrong and seemingly illogical adaptors. How to develop an internal representation for adaptors that fits a developers mind remains a topic for future work.

- An ideal test scenario should cover all the actions that will be invoked upon the server in all possible combinations. How can we write good tests that do not leave any open holes for the programmer? And if we can write such tests, are the Petri nets still necessary? In other words, is it possible we learn the interface adaptor automatically just by looking at the interaction between the agents?

- Some interface conflicts that seem simple at first sight cannot be solved (not even by humans). For example, in our case we used a locking and unlocking strategy that can lock simple (x,y) specified cells on a board. It is impossible to interface this with a locking strategy that locks and unlocks the whole board at once. A simple solution such

as 'lock all fields' will not work because other communication partners can enter the field and lock a single position. This indicates that the approach presented here is good in solving 'control flow' problems but is bad at converting 'data representations'. However, this is a general AI problem [Mor96] for which no solution yet has been found.

- One can wonder whether the genetic algorithm will always generate a solution. In practice, if there is an adaptor, an adaptor will be found, because the mutation rate of the genetic algorithm is rather high. This can be compared to the energy by which a fitness landscape is searched. The high mutation rate guarantees that all points in the energy landscape are covered while the cross-overs guarantee that good looking solutions will be investigated in more detail. The solution that will be found will not always be the best solution, however. It will cover the basic requirements set by both test sets, but in may or may not optimise the required behaviour. If we want to take advantage of this we need to refine the fitness measure. This new fitness measure should be able to say wether an adaptor is *good enough* but also *how good* exactly.

In this paper we presented only a simple example with small interfaces (binary versus counting semaphore locking strategies) for the sake of the presentation. Currently we are in the middle of some very promising and far from trivial experiments that map one transaction interface (lock, unlock, commit, abort, where we cannot commit or abort unless all locks are released) to another similar transaction interface (lock, unlock, commit, abort where we can commit or abort when not all locks are released).

6 Conclusion

We proposed an automated approach to create intelligent interface adaptors to make mobile multi agents with incompatible interfaces communicate. Such an approach is indispensable in the emerging paradigm of peer-to-peer computing to cope with the combinatorial explosion of interface adaptors that are needed in an open distributed setting where agents migrate and interact with other agent systems in unpredictable ways.

Our approach uses a genetic algorithm that learns a classifier system. This classifier system contains classifiers which react upon the context they receive from both client agent and server agent. The context is defined as a combination of the possible client-side and server-side states as given by a user-specified Petri net. To measure the fitness of a solution, the user needs to provide test scenarios as input. This enables the user to avoid undesired behaviour in the agent interactions. As a result, the responsibility of the designer of a mobile agent system is shifted from writing correct interface adaptors to specifying correct test scenarios.

As an initial experiment to validate our approach we used concurrency because this is known to be a major problem in peer-to-peer computing. More specifically, we automatically generated an interface adaptor between incompatible locking strategies. Obviously, the approach is not restricted to the domain of concurrency, but more elaborate experiments need to be carried out. This includes using larger interfaces (with more functions, hence larger Petri nets).

Acknowledgements

Thanks to Johan Fabry, Tom Lenaerts, Anne Defaweux and Tom Tourwé for proofreading this paper.

References

[BF93] L. Bull and T. Fogarty. Co-evolving communicating classifier systems for tracking. *Proc. Int'l Conf. Neural Networks and Genetic Algoriths*, 1993.

[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.

[BHRS98] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole - concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, 1998.

[CDL+96] K. M. Chandy, B. Dimitrov, H. Le, J. Mandleson, M. Richardson, A. Rifkin, A. G. Sivilotti, W. Tanaka, and L. Weisman. A world-wide distributed system using java and the internet. *Proc. Int'l Symposium on High Performance Distributed Computing*, 1996.

[Glo89] David E. Glodberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

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

[KvdS96] B. Kröse and P. van der Smagt. *An introduction to neural networks*. University of Amsterdam, November 1996.

[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.

[Mor96] L. Morgenstern. The problem with solutions to the frame problem. In K. M. Ford and Z. Pylyshyn, editors, *The Robot's Dilemma Revisited: The Frame Problem in Artificial Intelligence*, pages 99–133. Ablex Publishing Co., Norwood, New Jersey, 1996.

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

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning – An Introduction*. MIT Press, 1998.

[SDD99] C. Sanza, C. Destruel, and Y. Duthen. Autonomous actors in an interactive real-time environment. *Proc. ICVC'99, Goa, India*, February 1999.

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

[Som97] B. Sommers. Agents: Not just for bond anymore. *JavaWorld*, April 1997.