THE CBORG MOBILE MULTI-AGENT SYSTEM

Werner Van Belle
Vrije Universiteit Brussel
Pleinlaan 2
1050-B Brussels
+32 2 629 34 81
werner.van.belle@vub.ac.be

Karsten Verelst
Vrije Universiteit Brussel
Pleinlaan 2
1050-B Brussels
+32 2 629 34 81
kaverels@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel
Pleinlaan 2
1050-B Brussels
+32 2 629 34 80
tjdhondt@vub.ac.be

ABSTRACT
This paper describes the design behind an experimental mobile multi agent system we have built over the past 4 years. The architecture itself is called Cborg. The objective was to create an architecture which can be easily used to implement intelligent autonomous agents in a wide area network.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs.

General Terms
Design, Experimentation, Languages, Theory.

Keywords
Strong mobility, distribution, mobile multi agent systems, virtual machines.

1. INTRODUCTION
The original definition of Pattie Maes [], states that an agent is an interactive entity which represents its owner on the net, eventually negotiating with other agents on his/her behalf. This visionary definition of intelligent agents has been proven to work out in Firefly []. Nevertheless, we still lacked good architectural support for these kind of agents. Writing agents was (and is) still hard labour, with more design going into the agent architecture than in the agents themselves. As such, mobile multi-agent systems were developed at various research labs. The CBorg mobile multi agent system is one of these systems. Its design as a distributed architecture in which agents can communicate and migrate allows easier implementation of intelligent agents.

From our point of view, 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. The mobile multi-agent system is the architecture which allows these kind of agents. A mobile multi-agent system should offer as much flexibility as possible to the agent programmer while hiding the underlying network architecture as much as possible.

The architecture we supply aims at programmers of intelligent autonomous agents. The agent system takes care of all kind of nasty stuff which the programmer doesn't want to be aware. CBorg is presented to him/her as a combination of a language and an integrated development environment. CBorg aimes at wide area distributed environments.

2. ARCHITECTURE
Cborg agents ('Migrobs') are written in a high level lisp/scheme like programming language, called Pico. Pico is an educational language used at the 'Vrije Universiteit Brussel' to teach first grade computer scientists concepts of programming languages. Small and portable by design, Pico allows us to experiment with new concepts and language constructions for mobile multi-agents. As such, we don't have any intention to be compatible with existing languages, such as Java.

On top of Pico, we have an integrated development environment (IDE) which supports the interactive writing and designing agents.

If we start the Cborg interpreter, it connects to another Cborg interpreter to give an interconnected network of agent systems. This interconnection of mobile multi agent systems is called the mobile multi agent infrastructure.

Currently, the CBorg mobile multi-agent architecure features

- **Strong mobility** meaning that an agent can migrate between agent systems in the middle of its execution. Strong mobility is seldomly found in current agent systems due to some technical drawbacks of Java. Because CBorg has the ability to reify the complete computational state of a running process, including its runtime stack, strong mobility is one of its standard features. Furthermore we have the ability to store and retrieve computations as variables (**continuations**) and pass this to other agents (**remote continuations**). []
- An easy to use standard **communication** layer between agents. This communication layer which consists of a serialisator and an objectcall-like syntax allows agents to pass objects to each other. Agents communicate always in an asynchrounous fashion. The reasoning behind this design decision is the notion of being 'autonomous': an atonoumous agent should be designed as a separate entity, sending messages to other agents, not as an entity which transfers its control flow to other agents.
- **An hierarchical naming system**: every agent has a human-readable name which is always used to reference it. This naming system favors late binding, in the sense that we bind agents to each other at execution time, not at compile, as we partially do with objects.
- **High performant location transparent messaging system**: an agent can send messages to another without having to know where the other agent resides. Eg: if agent 'Allice' talks to agent 'Bob', and 'Bob' migrates to the agent system at the end of the universe, CBorg will keep on routing messages

between Bob and Allice using the shortest path between them [].

- **Resource Transparency**: all resources in the mobile multi-agent system (disks, user interfaces and so on) are represented as static agents (which cannot migrate). So whenever we migrate an intelligent agent it stays connected to the resources it was using.
- **Garbage Collection**: as in every good language we have a state of the art, highly performant, garbage collector incorportated in the system. The garbage collector uses a 32 bit cell memory, with only 2 bits reserved for bookkeeping !
- **Synchronisation between agents** is done by using a rendez-vous between multiple agents. This rendez-vous can be in time and/or in space (synchronize at a certain place). The primitives themselves are based upon CSP, with the exception that we use unification as guards instead of sequenced statements.

All these features will be illustrated in section 3 (Using Cborg), while the internal workings and the design behind the system is explained in section 4 (Internals)

3. USING CBORG

3.1 The Basics

Cborg is dynamically typed and a variable can contain values of the following type: numbers, text, tables, functions and references to agents.

a:5 *(definition)*
->5
a:=10 *(assignment)*
->10

Equivalent to variables, we can define tables and assign values to any cell of the table.[1]

```
a[5]:0
->[0 0 0 0 0]
a[3]:=5
->[0 0 5 0 0]
```

Functions can also be defined using an equivalent intuitive syntax.

```
average(a,b):
        {sum:a+b;
        avg:sum/2;
        avg}
-><function average>
average(4,10)
->7
```

This is about the entire abstract grammar we need for our agent-language. Allmost all other specialities are handled by the native library. This includes control flow primitives such as *if*, *do*, *until*; aritmethic functions, relational functions, string-processing functions, input/output functions and others.[2]

[1] Contrary to Lisp or Scheme, everything in pico is stored in tables, not in pairs.

[2] For people interested in a nice feature of Pico: we can write a functionheader as function(delay())::<body>. When we call this function the actual argument will not be evaluated but assigned to delay(), which can afterwards be evaluated to obtain the result. This is called *delayed evaluation*.

3.2 Creating Objects

Pico semantics were extended to support objects. The result is a prototype-based language. Objects can specified and cloned in a very simple but effective way. In the transcript below `makecircle` denotes a mixin method which extends the basic point to become a circle:

```
createpoint(x,y)::
  { getx():: x;
    gety():: y;
    setx(nx):: x:=nx;
    sety(ny):: y:=ny;
    showx()::display(x);
    showy()::display(y);
    makecircle(r)::
         { setr(nr):: r:=nr;
           getr():: r;
           clone() };
    clone() }
:<closure createpoint>
```

We can now create a point by calling a:createpoint(1,2); if we want to make a circle from this point we call b:a.makecircle(900). Retrieveing values is done by calling getx(). For example b.getx.

The code above contains two kind of definition statements. The first (called definition) is annotated with a colon (:), while the second one (called declaration) is annotated with a double colon, or quad-point (::). Defined statements will, whenever the dictionary is copied, be copied by value, while declared statements will be copied by reference. In the code above, this means that all the declared functions are copied by reference (there code remains the same for all objects), while the field x and y, are copied by value.

3.3 Naming

Current naming schemes in mobile multi-agent systems are mainly based on internet naming, without taking into account the possible mobility of agents. If an agent migrates to another host its name changes to point out its new location. For example, if the agent progwww.vub.ac.be/hello moves to another agent system progpc15.vub.ac.be, its name will change to progpc15.vub.ac.be/hello. This changing of names upon migration is highly undesirable. In a mobile multi-agent system, every agent should have its own name, which (1) defines the agent in a unique fashion and (2) doesn't change whenever the agent migrates to another agent system. These naming restrictions make things for agents more complex than in current-day distributed systems.

In Cborg all agents do have a name which combines the name of the agent system (say, the computer) and a free form string which is used to 'name' the agent. For example we have an agent system tecra.vub.ac.be which creates an agent ralf. This agent will be called tecra.vub.ac.be/ralf. To avoid name clashes we say that every agent system manages its own name space. So, machine tecra.vub.ac.be shouldn't spawn more than one tecra.vub.ac.be/ralf agent.[3]

This agent name tecra.vub.ac.be/ralf is used as a human readable unique identifier which is also used to reference it when we want to send a messages to it. Referneces are made with the agentref primitive:

[3] Of course, if agents are also able to spawn other agents, it may be useful to change the naming system to something like 'creationmachine/ownermachine/<name>'

```
a:agentref("tecra/alice")
```

3.4 Creating a Migrob

CBorg has 2 ways to create Migrobs. The first one is the easiest: just click on the 'new'- button. If you do this, 2 agents will be created. The first one, invisible to the user is the actual, computaing agent, and a second one which is visible and the user interface agent.

The second way to create an agent is by calling the agentclone primitive.

3.4.1 AgentClone

Agentclone is a cloning operator which copies a complete environment and puts this into a new Migrob. This way, variables and functions can be passed on to the child. Note that a copy is made of all functions and variables, and that therefore no concurrency problems can occur between parent and child.

For example, if we look at the previous example where we wanted to transfer the point. We can now encapsulate the point and migrate it to the receiver.

```
a:createpoint(10,20);
b:agentclone(a,"punt");
c:agentref("Tecra/Bob");
b.migrate(placeof(c));
c.show(b);
```

3.4.2 NewAgent

If we are tired of writing objects ourselves we can easily encapsulate a function by using *newagent. Newagent* creates a new agent that runs independently and concurrently with it's creator. For this, newagent must know the name for this new agent.

foo(a,b):*<some_difficult_computation(a,b)>;*
newagent('helper',foo(5,8));

3.5 Messages between Migrobs

Cborg has a mean to send messages from one agent to another, in a fairly obvious way. Resembling Java RMI easiness, we refer to a remote agent using its name, and afterwards we use this reference just as we would send a message to an object. For Example:

```
a:agentref("Tecra/Christine")
a.display("Hello")
```

Sending messages this way ensures the *asynchronous* delivery of a message from one agent to another. Because the sending command is asynchronous, execution of this instruction results in putting the message in the message delivery system. The call returns immediately <void> and continues. This includes the sender doesn't wait for the message to actually arrive, speeding up program execution by avoiding large delays that can be possible in wide area networks. The downside of asynchronous communication is of course that errors are difficult to detect.

All parameters passed to a remote procedure are serialized automatically. Nevertheless all objects which are locally remains locally and are passed as a reference. For example, if we have a point created locally and we pass this to another machine who calls 'show' on that point. The point will actually be shown on the senders machine, not on the receivers.

```
Tecra/Alice
a:createpoint(10,20);
b:agentref("Tecra/Bob");
b.show(a)

Tecra/Bob
show(p)::p.show()
```

This maybe strange feature allows easy implementation of callback functions and object as is illustrated in the next small example. The first agent (Alice) is the receiving agent which will do a callback to the second agent (Bob). After creating a callback procedure Bob calls Alice and receives the answer in this special made ''object''.[4]

```
Tecra/Alice
Calculate(…,callback)::
  { <some calculation>;
    callback.Answeris(…) }

Tecra/Bob
Answeris(…)::
  { display("The answer is: ");
    display(result) };
agent: remotedict("Tecra/Alice");
agent.Calculate(…,agentself())}
```

3.6 Synchronisation

If we ever want to be able to synchronize processes in these kind of systems we need a good underlying fundamental communication model which includes synchronisation between processes. We can think of CSP, pi-calculus, actor systems but it turns out that all these models are only valid in small scaled systems.

When using agents, we need a synchronisation mechanism, that allows to synchronize between processes of different owners, to synchronize in a network with unpredictable delays, to communicate data to other processes

3.6.1 Sync

A primitive we've added to our language is called sync and synchronizes two or more agents.

Table 1. Sync behaviour

Bob	Alice
sync("Alice",a,10)	…
…. Waiting for sync	…

[4] In this example we see that this way of working has a number of advantages over standard distributed systems:

- We don't have to generate and compile stubs beforehand. (in comparison to Java and CORBA this is definitely a strong advantage, particularly when we transfer code to other locations)
- The code is interpreted which guarantees small and powerful pieces of code (compiled code is much larger than interpreted code)
- We don't have to take location of an agent into account, since an agent's name doesn't change after migration.
- Sending messages between agents is similar to sending messages between objects, with automatic serialization of messages.

….	
…	Sync("Bob",20,b)
Execution Continues with a=20	Execution Continues with b=10

The parameters for the sync primitive are equivalent to that of the receive. First of all the other agent we wish to sync with must be specified. Again this can be done by explicitly naming the other agent, using a wildcard, or it could be a table with all possible synchronees. The second optional argument is a pattern, putting restrictions on what agents may synchronize. This way we can choose our synchronee with other means than it's name.

3.6.2 Synchronous Messages

Combining the send, receive and sync primitive, we can simulate a synchronous send and receive. For example: if Alice wants to send 'hey' to Bob and wants to be sure Bob has received it, it can state something like:

```
Bob.hey();
sync(Bob,"hey")
```

of course, Bob should respond with sync(Alice,"hey") after receiving and handling 'hey'. This kind of synchronous send and receive are implemented in the language.

```
srecv(from,pattern):
{sync(from,pattern);
recv(from,pattern)}
```

```
ssend(to,message):
{send(to,message);
sync(to,message)}
```

3.7 Example 1: Http Daemon

The exzample below is an illustration how one can use agents in an existing infrastructure. The setup consists out of a number of agents roaming the web (if they want) while acting as a cgi script towards a http daemon. This Http daemon allows subscribers and redirect all incoming webpage requests towards the subscribed agent, which in turn generates a homepage.

```
WebRequest: this agent is created by the C
interface for every incoming request.
display("<html>");
handlehtmlrequest(brol)::
        {
        a:remotedict("tecra/webdispatcher");
        a.handlehtmlrequest(agentself(),brol)
        };
finish()::
        {
        display("</HTML>");
        agentdie()
        }


WebDispatcher
handlers[100]:["",0];
findhandler(line)::
        <…>
respondto(req,line,formdata)::
        {
        req.display("<h1>Unknown URL:"
        req.display("<BR></H1><H3>");
        req.display(line);
        req.display("</h3>");
        req.finish()
        };
createformdata(line)::
        <…>
handlehtmlrequest(req,line)::
        {h:findhandler(line);
        formdata:createformdata(line);
        h.respondto(req,line,formdata)};
forgethandler(handler)::
        <…>
announcehandler(handler,agent)::
        <…>
```

3.8 Example 2: Transactions

This example shows how we can implement a simple transaction system in Cborg.

```
ctxnr:1;
nolock:0;
islock:1;
tabelsize:5;

makeLockman()::
        {lock:nolock;
        owner:0;
        getOwner()::owner;
        canread(newowner)::
                if (lock = nolock,
                        {owner := newowner;
                        lock := islock;
                        true}
                ,       owner = newowner);
        canwrite(newowner)::
                if (lock = nolock,
                        {owner := newowner;
                        lock := islock;
                        true}
                ,       owner = newowner);
        release(oldowner)::
                {display(owner);
                if (oldowner = owner,
                        {owner := 0;
                        lock := nolock
                        },void)};
        clone()};

db[tabelsize]:["",void,makeLockman()];

findbinding(db,name):: <…>;

makeContext(target)::
        {id:ctxnr;
        ctxnr := ctxnr + 1;
        Read(name)::
                {binding:findbinding(db,name);
                lockman : binding[3];
                r:localReference();
                result: lockman.canread(id);
                if(result,
                        ssend(target,binding[2]),
                        r.Read(name))};
        Write(name,value)::
                {binding:findbinding(db,name);
                lockman : binding[3];
                r:localReference();
                result:lockman.canwrite(id);
                if(result,
                ssend(target,binding[2]:=value),
                r.Write(name,value))};
        Commit()::
                {i:1;
                while(i<tabelsize+1,
                        {binding:db[i];
                        temp:binding[3];
                        temp.release(id);
                        i := i + 1});
                ssend(target,true)};
```

```
        Abort()::Commit()
        showtxid()::display(id);
        clone()};

BeginTransaction(target)::
        {ctx:makeContext(target);
        ssend(target,ctx)};
```

4 INTERNAL WORKINGS

!! See Appendix 1 for an illustration !!

4.1 Cellmemory + Garbage colelctor

Everything the evaluator does is stored in a special cell-memory. This includes the processes and runtime stacks. This memory is garbage colelcted and allows easy serialisation of object, procedures, native types and agents.

The cellmemory itself contains words with length 32, with 1 bit reserved for bookkeeping by the garbage colelctor (!!!). Every cell is tagged with a number from 0 to 31., indicating the kind of expression it is. We have VOI (void), FRC (fractions), TXT (strings), CCC (callback functions to C),, RDC (agent references), THK (thunks), TAB (tables), FUN (functions), NAT (Natives), REF (Local References), APL (Applications), TBL (Table indexing), DEF (Definitions), DCL (Declarations), ASS (assignments), MES (message sends), SUP (super calls), DCT (dictionaries) , BND (bindings), ENV (environments/proifesses), and NBR (numbers)

4.2 Serialisation

We have installed a serializer to store and retrieve subgraphs of the data store. The serializer differentiates between two kinds of serialization. The first concerns expressions handled as messages sent to another location. The second concerns expressions handled as complete agents which should be sent to a remote host.

When a message is serialized, we traverse the data graph and store everything that we encounter on a stream (outside the cellmemory). So this process does take twice as much memory as the original message. The process stops at leaves and at dictionaries which will be serialized as references to remote dictionaries. In this way, an agent can send a local dictionary to another agent without having to send the entire dictionary content to the communication partner.

Serializing agents consists of traversing the data graph and storing everything we encounter, including local dictionaries. The one exception to this is the root environment which is simply marked as being the root environment. In this way we can integrate agents in their new agent environment upon their arrival at a new location.

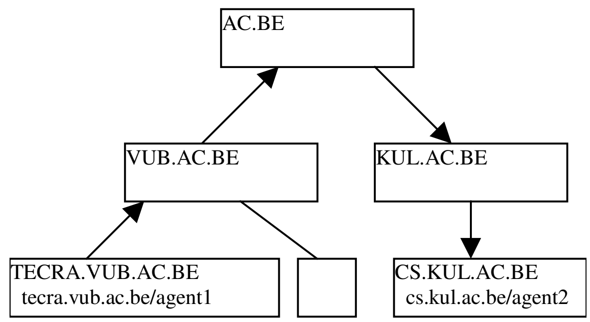
4.3 Location Transparant Routing

4.3.1 Sending Messages

The solution we propose is to make no distinction between the name of an agent and the address of an agent. Instead of resolving the name of an agent to find its place we immediately route messages to an agent based upon the receiver's name. This means of course that we need to change the infrastructure substantially. We no longer have a statically interconnected routing infrastructure and a different statically interconnected naming infrastructure, instead we have one hierarchical infrastructure in which we name and route messages.

Instead of sending a lookup request to a nameserver, we send the complete message to the nameserver, which will 'route' the messages further to the next nameserver.

For example: the figure below contains an hierarchical interconnection of nameservers/routers. If agent tecra.vub.ac.be/agent1 wants to send a message to an agent cs.kul.ac.be/agent1, he can pass the message to the local nameserver, in this case vub.ac.be. At the moment vub.ac.be receives a message it will send the message through to ac.be. Ac.be will see the message for cs.kul.ac.be and will pass it to the right downlink, in this case, kul.ac.be. From there on, it is delivered to cs.kul.ac.be/agent1 in exactly the same way.



More formally, the delivery of a message on every node is as follows:

1. Is the message for a name which is local ? Yes, deliver the name, No continue.
2. Is there a forward rule for this name ? Yes, forward the message to the given link, No continue
3. Is there a link with the same name suffix ? Yes, deliver the message to that link. No, continue
4. Deliver the name to the uplink.

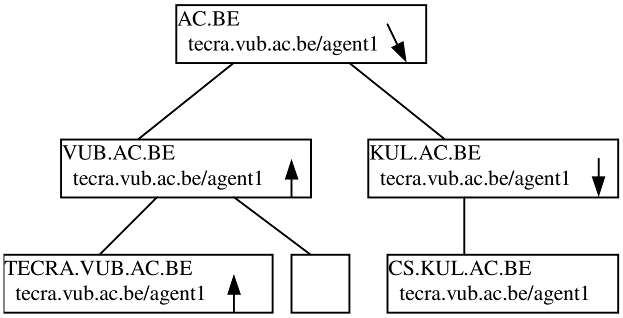
4.3.2. Migrating Agents

To migrate an agent in this setup, we do it in exactly the same way as if we would send a message, with the difference that every router/nameserver should *understand* the movement of an agent. This means that whenever an agent passes through a node, the node should update its routing-tables to point out the agent's new *direction*. Note that we do not point to the new location, but instead show the link to where the agent has gone to. This guarantees that there are no updates needed to this node, if the agent migrates further on.

For example, if we migrate tecra.vub.ac.be/agent1 to cs.kul.ac.be, we take the state of the agent and send it as an agent message to the local node tecra.vub.ac.be. This node will send the agent to its uplink, vub.ac.be and update its routing table to let tecra.vub.ac.be/agent1 point towards the uplink. Vub.ac.be will send the agent to ac.be and will keep a rule for agent tecra.vub.ac.be/agent1 to its uplink. The node ac.be will send the agent to the right downlink, in this case kul.ac.be and make a rule to point out the agent's new location.

If an agent moves through a node to its original position, the associated rules will be deleted. To keep tables smaller we can use a system of wildcards to annotate groups and clusters of agents.

If an agent migrates while messages are being sent to it, these messages will follow the agent on its path and will finally arrive at the right location.



More formally: Every node has a routing table which keeps track of migrated agents. The algorithm to deliver an agent is on every node as follows:

1. Is the agent for this machine. Yes, deliver the agent and update the local table. No, continue
2. Is the destination place name higher in the tree ? Yes, send the agent to the uplink and update the local table with a rule '<agent> -> uplink'. No, continue
3. Is there a downlink with a name that is a suffix for the given placename. Yes, send the agent to the given link and add a rule '<agent> -> downlink <x>'. No, continue
4. Send the agent back, destination place doesn't exist.

4. ACKNOWLEDGMENTS

We would like to thank the whole team who helped devloping this mobile multi agent system. Team members are Theo D'Hondt, Werner Van Belle, Karsten Verelst, Dirk Van Deun and Johan Fabry.

5. REFERENCES

[1] K. Rothermel and R. Popescu-Zeletin, Eds. *Mobile Agents,* Lecture Notes in Computer Science Series, vol 1219, Springer 1997

[2] R.S. Gray, *Agent TCL: A Transportable Agent System*, Proc. CIKM, Wksp Intilligent Information Agents, J.Mayfield and T. Finnin, Eds. 1995

[3] http://www.trl.ibm.co.jp/aglets: *The Aglet Workbench*

[4] http://www.objectspace.com/voyager: Voyager

[5] http://www.omg.org/… The Corba homegrounds. Here you can also find information about the Mobile Agent Facility

[6] http://www.chatsubo.com/rmi: Java RMI Development Page

[7] *J. Solomon* Mobile Ipv4 Configuration Option for PPP, IPCP, The internet Society RFC2290, Feb '98

[8] *Charles E. Perkins,* IP Mobility Support Sun Microsystems, Oct '96

[9] *Norman C. Hutchinson:* Emerald: An Object-Based Language for Distributed Programming; Departement of computer science, university of Washington, Seattle, Jan '87

[10] *C.A.R. Hoare,* Communicating Sequential Processes, Prentice Hall International Series in Computer Science, 1985. ISBN 0-13-153271-5 (0-13-153289-8 PBK).

[11] *Milner, Parrow and Walker*, A Calculus of Mobile Processes Pt.1 LFCS report ECS-LFCS-89-85

[12] *Milner, Parrow and Walker* A Calculus of Mobile Processes Pt.2 LFCS report ECS-LFCS-89-86

[13] *Robin Milner* The Polyadic pi-Calculus: A Tutorial LFCS report ECS-LFCS-91-180

[14] www.pico.vub.ac.be

[15] http://prog.vub.ac.be/poolresearch/CBorg/

[16] *Gul Agha, Ian A Mason, Scott F Smith, Carolyn Talcott,* a Foundation for Actor Computation, Cambridge University Press 1993

[17] Jim White, *Mobile Agents White Paper*, General Magic

[18] Robert S. Gray, *Agent TCL: A flexible and secure mobile-agent system*, Department of Computer Science, Dartmouth College

[19] Gian Pietro Picco, *Understanding Code Mobility*, Politecnico di Torino, Italy, Tutorial at ECOOP98, 22 July 1998

[20] Gerald Jay Sussman, Guy Lewis Steele Jr, *Scheme, An Interpreter for Extended Lambda Calculus*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory. December 1975

[21] Werner Van Belle, *Reinforcement Learning as a Routing Technique for Mobile Multi Agent Systems*, April 1997

[22] Ad Astra, *Jumping Beans, a white paper* , 1 september 1998

[23] Online documentation on http://pico.vub.ac.be

[24] Agha G. *Actors: A Model of Concurrent Computation in Distributed Systems*, AI Tech Report 844, MIT, 1985

APPENDIX I