Clustered Computing with NetLogo for the evaluation of asynchronous search techniques

Ionel Muscalagiu
The Faculty of Engineering of Hunedoara
The "Politehnica" University of Timisoara
Hunedoara, str. Revolutiei, nr.5, Romania
Email: ionel.muscalagiu@fih.upt.ro

Horia Emil Popa
The Faculty of Mathematics and Informatics
The University of the West, Timisoara
Timisoara, V.Parvan 4, Romania
Email: hpopa@info.uvt.ro

Jose Vidal
Computer Science and Engineering
University of South Carolina
Columbia SC 29208, USA
Email: vidal@sc.edu

***Abstract*—Distributed Constraint programming is a programming approach used to describe and solve large classes of problems such as searching, combinatorial and planning problems. A Distributed Constraint Satisfaction is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents. Modelling and simulation are essential tools in many areas of science and engineering, including computer science. The purpose of this article is to present an open-source tools in NetLogo for implementation and evaluation of the asynchronous searching techniques for a great number of agents, tool that can be run on a cluster of computers. Also, in this paper we have developed a methodology to run the NetLogo models in a cluster computing environment or on a single machine, varying both parameter values and/or random number of agents. Such a tool allows using various search techniques also the evaluation and analysis of performances of the techniques for the asynchronous searching techniques, the study of agents behaviour in several situations, like the priority order of the agents, the synchronous and asynchronous case, apparition of delays in message transmission, leading, therefore, to identifying possible enhancements of the performances of asynchronous search techniques.**

I. Introduction

Constraint programming is a programming approach used to describe and solve large classes of problems such as searching, combinatorial and planning problems. A Distributed Constraint Satisfaction Problem (DisCSP) is a constraint satisfaction problem in which variables and constraints are distributed among multiple agents citeyoko, [6]. This type of distributed modelling appeared naturally for many problems for which the information was distributed to many agents.

The idea of sharing various parts of the problem among agents that act independently and collaborate in order to find a solution by using messages has led to the formal problem known as the Distributed Constraint Satisfaction Problem (DisCSP) [11], [6]. DisCSPs are composed of agents, each owning its local constraint network. Variables in different agents are connected by constraints forming a network of constraints. Agents must assign values to their variables so that all constraints between agents are satisfied. Distributed networks of constraints have proven their success in modelling real problems. There are complete asynchronous searching techniques for solving the DisCSP in this constraints network, such as the ABT (Asynchronous Backtracking), AWCS (Asynchronous Weak Commitment), ABTDO (Dynamic Ordering for Asynchronous Backtracking), DisDB (Distributed Dynamic Backtracking) and DBS (Distributed Backtracking with Sessions) [1], [2], [6],[11].

The asynchronous search techniques, existent for the DisCSP modelling, are within the framework of distributed programming. The agents can be processes residing on a single computer or on several computers, distributed within a network. The implementation of any asynchronous search techniques supposes building the agents and the existing constraints, the implementation of the links between the agents and the communication channels between them. The implementation of asynchronous search techniques can be done in any programming language allowing a distributed programming, such as Java, C, C++ or other. Nevertheless, for the study of such techniques, for their analysis and evaluation, it is easier and more efficient to implement the techniques under a certain distributed environment, which offers such facilities (NetLogo [12], [16], [17], [9], [4]).

NetLogo [12] is regarded as one of the most complete and successful agent simulation platforms [9], [4]. NetLogo is a high-level platform, providing a simple yet powerful programming language, built-in graphical interfaces and the necessary experiment visualisation tools for quick development of simulation user interface. It offers a collection of complex modelling systems, developed in time. The models could give instructions to hundreds or thousands of independent agents which could all operate in parallel. It is a environment written entirely in Java, therefore it can be installed and activated on most of the important platforms. Although, excellent for "modelling social and emergent phenomena", i.e. agent based simulations that consist of a large number of reactive agents, it lacks the facilities to model easily more complex goal oriented agent behaviours.

Modelling and simulation is an essential tool in many areas of science and engineering, including in computer science, for example, for analysing the performances of asynchronous search techniques. Developing of high performance search techniques assumes a message flow as little as possible and a minimal local effort for evaluating the constraints. The asynchronous search techniques can be remarked by the existence of a very large number of elements that can be introduced, without affecting the completeness of the algorithm. For example, processing the message in packets or individual, storage or not the nogood messages, message filtering through different techniques that are not affecting the completeness, nogood learning and storing them for each value. Another situation is tied to the structure of the constraints graph that occurs for the evaluated problems. Unfortunately, each asynchronous running of the same problem can give different results. Each run is affected by delays appeared in message transmission. Thus, a correct evaluation assumes a large number of runs, with different data sets. The purpose of this article is to offer a solution of implementation and evaluation for the asynchronous searching techniques, a way to automatize this process.

Developing of evaluation and testing tools for the search techniques became a necessity. There are very few platforms for implementing and solving DisCSP problems: DisChoco [13], DCOPolis [8] and FRODO. In this paper we have developed a methodology to run the NetLogo models in a cluster computing environment or on a single machine, varying both parameter values and/or random number of agents. We utilize the Java API of NetLogo as well as LoadLeveler. LoadLeveler is a job scheduler written by IBM, to control scheduling of batch jobs. LoadLeveler matches the job requirements with the best available computer resource for execution. It was primarily available only for AIX operating system, however it is now also available for IBM POWER and X86 architecture Linux systems.

The purpose of this article is to present an open-source solution for implementation and evaluation of the asynchronous search techniques in NetLogo, for a great number of agents, model that can be run on a cluster of computers. Such a tool allows the use of various search techniques so that we can decide which is the most suitable one for that particular problem. However, this model can be used in the study of agents behaviour in several situations, like the priority order of the agents, the synchronous and asynchronous case, leading, therefore, to identifying possible enhancements of the performances of asynchronous search techniques. We extend the model in [7] with support for running on a cluster of computers.

II. The Distributed Constraint Satisfaction Problem

This paragraph presents some notions related to the DisCSP modelling [11]. The Distributed Constraint Satisfaction Problem (DisCSP) has been formalized in [11], [6].

Definition 1: The model based on constraints CSP - Constraint Satisfaction Problem, existing for centralized architectures, is defined by a triple (*X, D, C*), where: *X*=$\{x_1,...,x_n\}$ is a set of n variables; whose values are taken from finite domains *D*= $\{D_1, D_2,...,D_n\}$; *C* is a set of constraints declaring those combinations of values which are acceptable for variables.

The solution of a CSP implies to find an association of values for all the variables that satisfies all the constraints.

Definition 2: A problem of satisfying the distributed constraints (DisCSP) is a CSP, in which the variables and constraints are distributed among autonomous agents that communicate by exchanging messages. Formally, DisCSP is defined by a 5-tuple (*X, D, C, A, ϕ*), where *X, D* and *C* are as before, $A = \{A_1,...,A_p\}$ is a set of p agents, and $\phi : X \longrightarrow A$ is a function that maps each variable to its agent.

The agent is responsible for setting the value of its own variable. The agents do not know the values of any other variable but can communicate with other agents. In this article we will consider that each agent A_i has allocated a single variable x_i, thus $p = n$. Also, we assume the following communication model [11]:

- agents communicate by sending messages. An agent can send messages to other agents iff the agent knows the addresses of the agents.
- the delay in delivering a message is finite, although random. For transmission between any pair of agents, messages are received in the order in which they were sent.

Asynchronous search algorithms are characterized by the agents using the messages during the process of searching the solution. Typically, it uses two types of messages:

- the *ok* message, which contains an assignment variable-value and is sent by an agent to the constraint-evaluating-agent in order to see if the value is right.
- the *nogood* message, which contains a list (called nogood) with the assignments wherefore a looseness was found, is sent in case the constraint-evaluating-agent finds an unfulfilled constraint.

III. Modelling and implementing of the asynchronous search techniques in NetLogo

In this section we present a solution of modelling and implementation for the existing agents' process of execution in the case of the asynchronous search techniques. This open-source solution, called DisCSP-NetLogo will be extended to be able to run on a larger number of agents, model runnable on a cluster of computers. This modelling can also be used for any of the asynchronous search techniques, such as those from the AWCS family [11], ABT family [1], DisDB [1], DBS [2]. Implementation examples for these techniques can be found on the sites in [16],[17]. This modelling solution in NetLogo was first presented in [7].

The modelling of the agents' execution process will be structured on two levels, corresponding to the two stages of implementation [7], [14]. The definition of the way in which asynchronous techniques will be programmed so that the agents will run concurrently and asynchronously will be the internal level of the model. The second level refers to the way of representing the NetLogo application. This is the exterior level. The first aspect will be treated and represented using turtle type objects. The second aspect refers to the way of interacting with the user, the user interface. Regarding that aspect, NetLogo offers patch type objects and various graphical controls.

A. Agents' simulation and initialization

First of all, the agents are represented by the breed type objects (those are of the turtles type). In fig. 1 is presented the way the agents are defined together with the global data structures proprietary to the agents. We implement in open-source NetLogo the agents' process of execution in the case of the asynchronous search techniques [7],[14]:

S1. Agents' simulation and initialization in DisCSP-NetLogo. First of all, the agents are represented by the breed type objects (those are of the turtles type). Fig. 1 shows the way the agents are defined together with the global data structures proprietary to the agents.

```
breeds [agents]
globals[variables that simulate the memory shared by all the agents]
agent-own [message-queue current-view MyValue nogoods
nr-constraintc messages-received-ok messages-received-nogood AgentC-Cost]
;message-queue contains the received messages.
;current-view is a list indexed on the agent's number, of the form [v0 v1...],
;vi = -1 if we don't know the value of that agent.
;nogoods is the list of inconsistent positions [0 1 1 0 ... ]
where 0 is a good position, and 1 is inconsistent.
;messages-received-ok and messages-received-nogood count
the n;umber of messages received by an agent.
; nr-cycles -the number of cycles
; nr-constraintc - the number of constraints checked
; AgentC-Cost - a number of non-concurrent constraint checks
```

Fig. 1. Agents' definition in DisCSP-Netlogo for the asynchronous search techniques

This type of simulation can be applied for different problems used at evaluation and testing:

- the distributed problem of the n queens, characterized by the number of queens (constant density for the constraints graph equal to $n*(n-1)/2$).
- the distributed problem of the m-colouring of a randomly generated graph, characterized by the number of nodes/agents, k=3 colors and the m-number of connections between the nodes/agents. Two types of graphs are defined: graphs with few connections (known as sparse problems, having `m=n x 2` connections) and graphs with a special number of connections (known as dense problems, `m=n x 2.7`).
- The randomly generated (binary) CSPs are characterized by the 4-tuple (n, m,p1,p2), where: n is the number of variables; m is the uniform domain size; $p1$ is the portion of the $n*(n-1)/2$ possible constraints in the constraint graph; $p2$ is the portion of the $m*m$ value pairs in each constraint that are disallowed by the constraint. That is, $p1$ may be thought of as the density of the constraint graph, and $p2$ as the tightness of constraints.
- The multi-robot exploration problem are are characterized by the 6-tuple (n, m, p1, sr, cr, obsd), where: - n is the number of robots exploring an environment, interacts and communicates with its spatial neighbours and sharing a few common information (information about already explored areas); - $m = 8$ is domain size of each variable; Dom(x_i)is the set of all 8 cardinal directions that a robot A_i can choose to plan its next movement. - $p1$ - network-connectivity - sr - the sensor range of a robot; - cr - the communication range of a robot. - $obsd$ - obstacles-density

S2.Representation and manipulation of the messages. Any asynchronous search technique is based on the use by the agents of some messages for communicating various information needed for obtaining the solution. The agents' communication is done according to the communication model introduced in [11].

The communication model existing in the DisCSP frame supposes first of all the existence of some channels for communication, of the FIFO type, that can store the messages received by each agent. The way of representation of the main messages is presented as follows:

- ◦ (list "type message" *contents* Agent-costs) ;

Examples:

- ◦ (list "ok" agent value agent-costs) - messages of the ok type; value= column of queen, color of nodes, cardinal directions.
- ◦ (list "nogood" agent current-view agent-costs) - messages of the nogood type.
- ◦ (list "addl" $agent_1$ $agent_2$ agent-costs)- add new link between $agent_1$ $agent_2$.

The simulation of the message queues for each agent can be done using Netlogo lists, for whom we define treatment routines corresponding to the FIFO principles. These data structures are defined in the same time with the definition of the agents. In the proposed implementations from this paper, that structure will be called message-queue. This structure proper to each agent will contain all the messages received by that agent.

The manipulation of these channels can be managed by a central agent (which in NetLogo is called observer) or by the agents themselves. In this purpose we propose the building of a procedure called *go* for global manipulation of the message channels. It will also have a role in detecting the termination of the asynchronous search techniques' execution. That *go* procedure is some kind of a "main program", a command centre for agents. The procedure should also allow the management of the messages that are transmitted by the agents. The procedure needs to call for each agent another procedure which will treat each message according to its type. This procedure will be called handle-message, and will be used to handle messages specific to each asynchronous search technique.

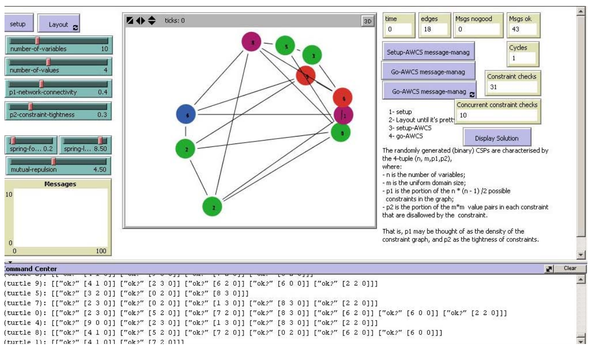
S3.Definition and representation of the user interface.

As concerning the interface part, it can be used for the graphical representation of the DisCSP problem's objects (agents, nodes, quens, robots, obstacle, link, etc) of the patch type. It is recommended to create an initialization procedure for the display surface where the agents' values will be displayed.

To model the surface of the application are used objects of the patches type. Depending on the significance of those agents, they are represented on the NetLogo surface. In figure 2 is presented the way in NetLogo for representing the agents.

S4.Running the application.

The initialization of the application supposes the building of agents and of the working surface for them. Usually are initialized the working context of the agent, the message queues, the variables that count the effort carried out by the agent. In figure 3 are presented the two routines of the

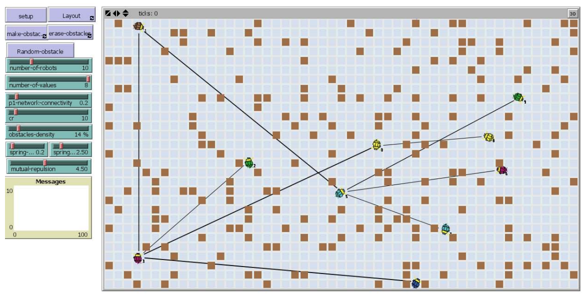


The randomly generated (binary) CSPs are characterised by the 4-tuple (n, m,p1,p2),
where:
- n is the number of variables;
- m is the uniform domain size;
- p1 is the portion of the n * (n - 1) /2 possible constraints in the graph;
- p2 is the portion of the m*m value pairs in each constraint that are disallowed by the constraint.

That is, p1 may be thought of as the density of the constraint graph, and p2 as the tightness of constraints.

```
(turtle 9): [["ok?" [4 1 0]] ["ok?" [2 3 0]] ["ok?" [6 2 0]] ["ok?" [6 0 0]] ["ok?" [2 2 0]]]
(turtle 5): [["ok?" [3 2 0]] ["ok?" [0 2 0]] ["ok?" [8 3 0]]]
(turtle 7): [["ok?" [2 3 0]] ["ok?" [0 2 0]] ["ok?" [1 3 0]] ["ok?" [8 3 0]] ["ok?" [2 2 0]]]
(turtle 0): [["ok?" [2 3 0]] ["ok?" [5 2 0]] ["ok?" [7 2 0]] ["ok?" [8 3 0]] ["ok?" [6 2 0]] ["ok?" [6 0 0]] ["ok?" [2 2 0]]]
(turtle 4): [["ok?" [9 0 0]] ["ok?" [2 3 0]] ["ok?" [1 3 0]] ["ok?" [8 3 0]] ["ok?" [2 2 0]]]
(turtle 8): [["ok?" [4 1 0]] ["ok?" [5 2 0]] ["ok?" [7 2 0]] ["ok?" [0 2 0]] ["ok?" [6 2 0]] ["ok?" [6 0 0]]]
(turtle 1): [["ok?" [4 1 0]] ["ok?" [7 2 0]]]
```

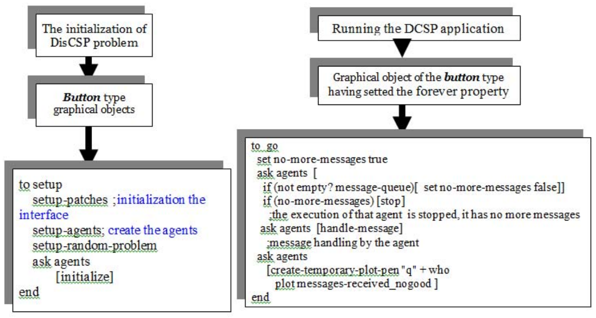
(a) The 2D square lattice representation for the graph coloring problem



(b) The 2D square lattice representation for the multi-robot exploration problem

Fig. 2. Representation of the environment in the case two problems

application initialization and running. The working surface of the application should contain NetLogo objects through whom the parameters of each problem could be controlled: the number of agents (nodes, robots), the density of the constraints graph. These objects allow the definition and monitoring of each problem parameters.

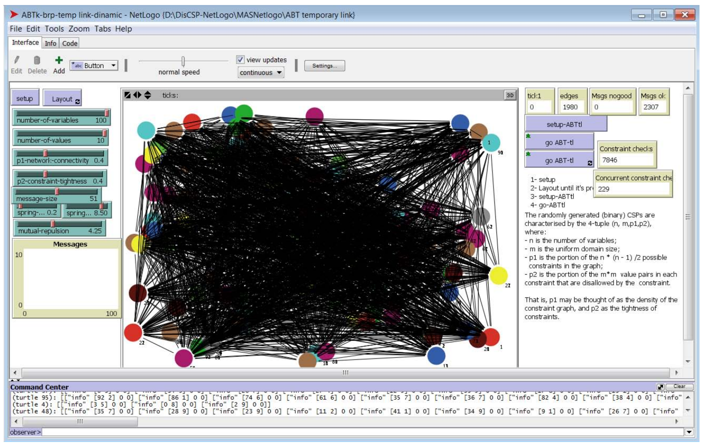

The initialization of DisCSP problem
Button type graphical objects
Running the DCSP application
Graphical object of the ***button*** type having setted the forever property

```
to setup
   setup-patches ;initialization the
interface
   setup-agents; create the agents
   setup-random-problem
   ask agents
        [initialize]
end
```

```
to  go
 set no-more-messages true
 ask agents  [
  if (not empty? message-queue)[  set no-more-messages false]]
  if (no-more-messages) [stop]
    ;the execution of that agent  is stopped, it has no more messages
  ask agents  [handle-message]
   ;message handling by the agent
 ask agents
   [create-temporary-plot-pen "q" + who
     plot messages-received_nogood ]
end
```

Fig. 3. Initialization and running of the DisCSP problem

For the application running there is proposed the introduction of a graphical object of the button type and setting the forever property. That way, the attached code, in the form of a NetLogo procedure (that is applied on each agent) will run continuously, until emptying the message queues and reaching the stop command. Another important observation is tied to attaching the graphical button to the observer [7]. The use of this solution allows obtaining a solution of implementation with synchronization of the agents' execution. In that case, the observer agent will be the one that will initiate the stopping of the DisCSP application execution (the update procedure or *go* procedure is attached and handled by the observer). These elements lead to the multi-agent system with synchronization of the agents execution. If it's desired to obtain a system with asynchronous operation, it will be used the second method of detection, which supposes another update routine [7], [16]. That new update routine will be attached to a graphical object of the button type which is attached and handled by the turtle type agents.



The randomly generated (binary) CSPs are characterised by the 4-tuple (n, m,p1,p2), where:
- n is the number of variables;
- m is the uniform domain size;
- p1 is the portion of the n * (n - 1) /2 possible constraints in the graph;
- p2 is the portion of the m*m value pairs in each constraint that are disallowed by the constraint.

That is, p1 may be thought of as the density of the constraint graph, and p2 as the tightness of constraints.

```
(turtle 95): [["info" [92 2] 0 0] ["info" [86 1] 0 0] ["info" [74 6] 0 0] ["info" [61 6] 0 0] ["info" [35 7] 0 0] ["info" [36 7] 0 0] ["info" [82 4] 0 0] ["info" [38 4] 0 0] ["info"
(turtle 4): [["info" [3 5] 0 0] ["info" [0 8] 0 0] ["info" [2 9] 0 0]]
(turtle 48): [["info" [35 7] 0 0] ["info" [28 9] 0 0] ["info" [23 9] 0 0] ["info" [11 2] 0 0] ["info" [41 1] 0 0] ["info" [34 9] 0 0] ["info" [9 1] 0 0] ["info" [26 7] 0 0] ["info"
```

Fig. 4. NetLogo implementation of the ABT with temporary links for the random binary problems, n=100 agents

In figure 4 is captured an implementation of the ABT with temporary links for the random binary problems technique that uses the model presented. The update procedure is attached and handled by the turtle type agents (figure 4). These elements lead to the multi-agent system with asynchronous agents operation. Implementation examples for the ABT family, DisDB, DBS and the AWCS family can be downloaded from the website [16].

IV. Running on a Linux cluster

In this paragraph we will present a methodology to run the proposed NetLogo models in a cluster computing environment or on a single machine. We utilize the Java API of NetLogo as well as LoadLeveler. LoadLeveler is a job scheduler written by IBM, to control scheduling of batch jobs. This solution is not restricted to operate only in this configuration depending on the workload management from the cluster. It can be used on any cluster with Java support.

Such a solution will allow running a large number of agents (nodes, variables, robots, queens, etc). The first tests allowed running of as much as 500 agents, in the conditions of a high density constraint graph. The first experiments were done on the InfraGrid cluster from the UVT HPC Centre [18], on 100 computing systems (an hybrid x86 and NVIDIA Tesla based). InfraGRID is an Linux only cluster based on a mixture of RedHat Enterprise Linux 6 and CentOS 6. For Workload Management, JOB execution is managed at the lowest level by IBM LoadLeveler.

The methodology proposed in the previous paragraph that uses the GUI interface will run on a single computer. In this paragraph we will propose a new solution, without GUI, that can run on a single computer or on a cluster.

The proposed solution uses the NetLogo model proposed previously, runnable without the GUI, with many modifications. In order to run the model that way is used a tool named BehaviourSpace, existent in NetLogo. BehaviorSpace is a software tool integrated with NetLogo that allows you to perform experiments with models in the ”headless” mode, that is, from the command line, without any graphical user interface (GUI). This is useful for automating runs on a single machine.

BehaviorSpace runs a model many times, systematically varying the model's settings and recording the results of each model's run. Using this tool we develop an experiment that can be runned on a single computer (with a small number of agent) or, in the headless mode on a cluster (using LoadLeveler as a job scheduler, with a large number of agents).

We will now present the methodology for creating such an experiment [5]. The steps necessary for the implementation of a multi-agent system are as follows:
S1. Create a NetLogo model according to the previous model for the asynchronous search techniques and for the types of problems used at the evaluation. For running it on the cluster and without GUI some adaptations have to be made. First, the NetLogo model must have a procedure called setup to instantiate the model and to prepare the output files. At a minimum it will need the following lines of code in fig 5.

```
to setup //Setup the model for a run, build a constraints graph.
  setup-globals //setup Global Variables
  setup-patches // initialize the work surface on which the agents move
  setup-turtles
    //we generate the objects of the turtles type that simulate the agents
  setup-random-problem
     // we generate a the types of problems used at the evaluation.
  setup-DisCSP
     // we initialize the data structures necessary for the DisCSP algorithm
end
```

Fig. 5. The Setup Procedure in DisCSP-Netlogo for the types of problems used at the evaluation.

Next, all models must also have a *go* (update) procedure. The *go* procedure is a bit different than the usual NetLogo program. The wrapper runs the NetLogo program by asking it to loop for a certain number of times and allows the finalizing of the DisCSP algorithm.

Usually for the DisCSP algorithms, the solution is generally detected only after a break period in sending messages (this means there is no message being transmitted, state called quiescence). This situation can be resolved by checking the message queues, queues that need to be empty. In such a procedure, that needs to run continuously (until emptying the message queues) for each agent, the message queue is verified (to detect a possible break in message transmitting).

The procedure should also allow the management of messages that are transmitted by the agents. The procedure needs to call for each agent another procedure (that will be called handle-message) and will be used to handle messages specific to each asynchronous search technique. The two procedures are the most important from the point of view of their messages handling way asynchronous or synchronous (way of work that defines the asynchronous techniques).

```
to go // The running procedure
    set no-more-messages true
    set nr-cycles nr-cycles + 1
    ask-concurrent agents [
      if (not empty? message-queue)[
        set no-more-messages false]]
    if (no-more-messages) [
      WriteSolution
    ]
    stop]
    ask-concurrent agents [handle-message]
end
```

Fig. 6. The Go Procedure in DisCSP-Netlogo for the asynchronous search techniques with synchronization of the agents' execution

The first solution of termination detection is based on some of the facilities of the NetLogo: the ask-concurrent command that allows the execution of the computations for each agent and the existence of the central observer agent. The handling of the communication channels will be realized by this central agent. These elements will lead to a variant of implementation in which the synchronizing of the agents' execution is done. This method allows obtaining a multi-agent system with synchronization of the agents' execution. Sample code for the *go* procedure (first solution) in the case of asynchronous search techniques can be found in fig 6.

S2. Create an experiment using BehaviorSpace and parse the NetLogo file into an input XML file (so that it can be runned in the headless mode, that is without GUI).

To finalize the run and adding up the results it is recommended the use of a Netlogo reporter and a routine that writes the results. The run stops if this reporter becomes true. In fig. 7 is presented a simple example of detecting the termination of execution for the asynchronous search techniques.

```
to-report Final
//reporter that will detect the termination of execution of the agents
    set no-more-messages true
    ask-concurrent agents [
      if (not empty? message-queue)[
        set no-more-messages false]]
    ifelse no-more-messages
      [ report true]
      [ report false]
end end
```

Fig. 7. The Final reporter in DisCSP-Netlogo for the asynchronous search techniques

In figure 8 there is presented this multi-agent system's architecture for running on a cluster of computers.

S3. Create a Linux shell script (in sh or in bash) that describes the job for LoadLeveler.

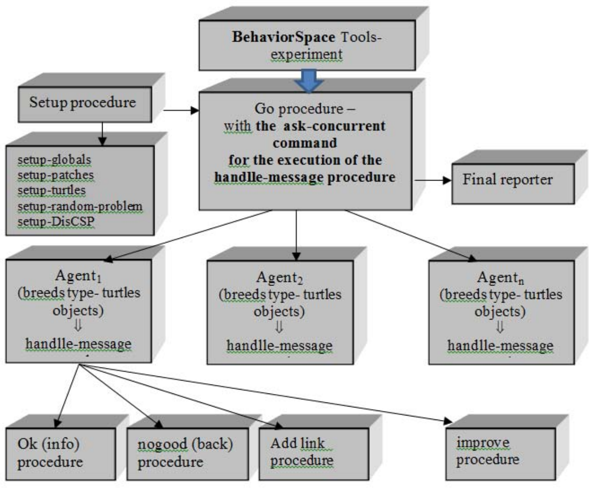

BehaviorSpace Tools-experiment
Setup procedure
setup-globals
setup-patches
setup-turtles
setup-random-problem
setup-DisCSP
Go procedure – with the ask-concurrent command for the execution of the handlle-message procedure
Final reporter
Agent1 (breeds type- turtles objects) ⇓ handlle-message
Agent2 (breeds type- turtles objects) ⇓ handlle-message
Agentn (breeds type- turtles objects) ⇓ handlle-message
Ok (info) procedure
nogood (back) procedure
Add link procedure
improve procedure

Fig. 8. Architecture of a multi-agent system with with synchronization of the agents' execution

Once the Netlogo model is completed with the experiment created with the BehaviourSpace tool, it is time to prepare the system for multiple runs. To do this, first create a new folder to use for file preparation and to collect the output created by the multiple runs. Place the NetLogo model in this folder.

Next create a script that allows running with no GUI: the process basically pulls the variables that are declared as sliders, choosers, and switches and then places them within XML tags. An example script is presented in fig. 9.

```
java −Xmx1024m // use up to 1GB RAM
-Dfile.encoding=UTF − 8 // for compatible cross-platform
-classpath NetLogo.jar // specify main jar
org.nlogo.headless.Main // specify that we want headless, not GUI
−−model NetLogo − Model.nlogo // the Netlogo model that runs
−−experiment name − of − experiment //the name of the experiment
```

Fig. 9. The script for multiple runs on a cluster for DisCSP

V. Conclusion

In this paper we introduce an model of study and evaluation for the asynchronous search techniques in NetLogo using the typical problems used for evaluation by extending the old model, model called DisCSP-NetLogo.

An open-source solution for implementation and evaluation of the asynchronous search techniques in NetLogo, for a great number of agents, model that can be run on a cluster of computers is presented. Such a tool allows the use of various search techniques so that we can decide on the most suitable one.

In this paper we have developed a methodology to run NetLogo models in a cluster computing environment or on a single machine, varying both parameter values and/or random number of robots. We utilize the Java API of NetLogo as well as LoadLeveler. The solution without GUI allows to be run on a cluster of computers in the mode with synchronization, as opposed to the GUI solution that can be runned on a single computer and allows running in both ways: with synchronization or completely asynchronously.

As a general conclusion, we think that the model will allow the use of the NetLogo environment as a basic simulator for the study of the the asynchronous search techniques. This model can run with synchronization (a synchronization of the agents' execution is done after each computing cycle) or agents process the received messages completely asynchronously.

Future research will include running more sets of experiments with two large families: the ABT family and the AWCS family applied to the typical evaluation problems (the distributed problem of the m-coloring of a randomly generated graph, the multi-robot exploration problem, the random binary CSPs) .

References

[1] C. Bessiere, I. Brito, A. Maestre,P. Meseguer, *Asynchronous Backtracking without Adding Links: A New Member in the ABT Family*. Artificial Intelligence, 161:7-24, 2005.

[2] Monier, P., Piechowiak, S. et Mandiau, R. (2009a). A complete algorithm for DisCSP: Distributed Backtracking with Sessions (DBS). In Second International Workshop on: Optimisation in Multi-Agent Systems (Opt-Mas), Eigth Joint Conference on Autonomous and Multi-Agent Systems (AAMAS 2009), Budapest, Hungary.

[3] Lytinen, S. L. and Railsback, S. F. The evolutionof agent-based simulation platforms: A review of NetLogo 5.0 and ReLogo. In Proceedings of the Fourth International Symposium on Agent-Based Modeling and Simulation, Vienna, Austria, 2012.

[4] Koehler, M., Tivnan, B., and Upton, S. Clustered Computing with NetLogo and Repast J: Beyond Chewing Gum and Duct Tape. Paper presented at the Agent2005 conference, Chicago, IL, 2005.

[5] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. *Comparing performance of distributed constraints processing algorithms*. Notes of the AAMAS'02 workshop on Distributed Constraint Reasoning, pages 86-93, Bologna, Italy, 2002.

[6] Muscalagiu, I., Jiang, H., Popa, H. E. *Implementation and evaluation model for the asynchronous techniques: from a synchronously distributed system to a asynchronous distributed system*. Proceedings of the 8th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC2006), Timisoara, IEEE Computer Society Press, 209–216, 2006.

[7] Sultanik, E.A., Lass, R.N., Regli,W.C.: Dcopolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In: AAMAS08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. (2008) 16671668.

[8] Railsback, S. F., Lytinen, S. L., and Jackson S. K. (2006). Agent-based simulation platforms: review and development recommendations. Simulation 82(9), 609-623.

[9] S. Tisue and U. Wilensky. Netlogo: Design and implementation of a multi-agent modeling environment. In Proceedings of the Agent 2004 Conference, 2004.

[10] M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara. *The distributed constraint satisfaction problem: formalization and algorithms*. IEEE Transactions on Knowledge and Data Engineering 10(5), page. 673-685, 1998.

[11] U. Wilensky. *NetLogo itself: NetLogo*. Available: http://ccl.northwestern.edu/netlogo/. Center for Connected Learning and Computer-Based Modeling, Northwestern University. Evanston, 1999.

[12] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere, El Houssine Bouyakhf. *DisChoco 2: A Platform for Distributed Constraint Reasoning*. Proceedings of the IJCAI11 Workshop on Distributed Constraint Reasoning (DCR11), pages 112–121, Barcelona, Spain, 2011.

[13] Jose Vidal. Fundamentals of Multiagent Systems with NetLogo Examples.Available: http://multiagent.com/p/fundamentals-of-multiagent-systems.html.

[14] *MAS NetLogo Models-a*. Available: http://discsp-netlogo.fih.upt.ro/.

[15] *MAS NetLogo Models-b*. Available:http://jmvidal.cse.sc.edu/netlogomas/.

[16] *MAS NetLogo Models-c*. Available: http://ccl.northwestern.edu/netlogo/models/community.

[17] *InfraGRID Cluster*. Available:http://hpc.uvt.ro/infrastructure/infragrid/