Simulating distributed algorithms for lattice agents

Oswin Aichholzer*[1], Thomas Hackl†[1], Vera Sacristán‡[2], Birgit Vogtenhuber*[1], and Reinhard Wallner[1]

[1]Institute for Software Technology, Graz University of Technology, Graz, Austria.
[2]Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Barcelona, Spain.

Abstract

We present a practical Java tool for simulating synchronized distributed algorithms on sets of 2- and 3-dimensional square/cubic lattice-based agents. This *AgentSystem* assumes that each agent is capable to change position in the lattice and that neighboring agents can attach and detach from each other. In addition, it assumes that each module has some constant size memory and computation capability, and can send/receive constant size messages to/from its neighbors. The system allows the user to define sets of agents and sets of rules and apply one to the other. The *AgentSystem* simulates the synchronized execution of the set of rules by all the modules, and can keep track of all actions made by the modules at each step, supporting consistency warnings and error checking. Our intention is to provide a useful tool for the researchers from geometric distributed algorithms.

Introduction

Mainly due to their scalability, distributed algorithms are a powerful tool for the control of self-organizing systems. One of the most interesting examples of a field of application is the control of modular robotic systems and, in particular, the development of geometric algorithms for their locomotion, reconfiguration, and self-repair. When dealing with these systems and algorithms, it is still often unaffordable to actually implement and run the algorithms on a big set of real prototypes and, in any case, it is recommended to simulate the behavior of the algorithms prior to their actual physical implementation. From a different viewpoint, frequently algorithmic results of theoretical nature are obtained but cannot immediately be translated into physical prototypes, as they may require miniaturization or precision to a level which is still out of reach. Having a simulator at hand is then very convenient. In this paper we present and describe the functionalities of a practical and very general simulator that we hope will be useful in many different research contexts. Other multi-agents simulators are available with different scopes, as for example MASON [5]. Our tool supplies a framework for lattice-based modular robots. The instruction set for the modules/agents is specific, simple, and intentionally compact, as to make the use easy also for non-experts. In the following descriptions we present the 2D information followed, if applicable, by additional information needed for 3D in squared brackets.

1 The agents

The initial agents setting is stored in the file `agents.txt`. Each line of the file defines one agent by its initial (global) coordinates (mandatory) plus (optional) its state, its attachments and initializations for (some of) its counters. Optionally it is possible to state the size of the universe in the agents file.

Universe size `UminX,maxX,minY,maxY[,minZ,maxZ]`
To be positioned at the beginning of the file.

Initial (global) position `x,y[,z]`
The initial position is written as integer x-, y- [and z-]coordinates, separated by a comma.

State `S_____`
The state of an agent consists of exactly 5 characters, written with a leading S.

Attachments `A____[__]`
The attachments of an agent are written as A followed by 4 [6] booleans (0 for not attached, 1 for attached), in the order north, west, east, south[, above, below].

Counters `C__ _____`
Each agent has 25 [45] integer counters, `C00,...,C24 [,C25,...,C44]`, which can be set to any 16-bit integer between -32767 and 32767.

*Email: {oaich,bvogt}@ist.tugraz.at. Research partially supported by ESF EUROCORES programme EuroGIGA - ComPoSe, Austrian Science Fund (FWF): I 648-N18.

†Email: thackl@ist.tugraz.at. Research supported by the Austrian Science Fund (FWF): P23629-N18 "Combinatorial Problems on Geometric Graphs".

‡Email: vera.sacristan@upc.edu. Research partially supported by projects MTM2012-30951, MTM2009-07242, Gen. Cat. DGR 2009SGR1040, and ESF EUROCORES programme EuroGIGA, CRP ComPoSe: MICINN Project EUI-EURC-2011-4306, for Spain.

2 The rules

The definition of what a robot may do is stored in the file `rules.txt`. Each rule definition consists of 4 lines:

1. the name of the rule,
2. the priority of the rule,
3. the precondition, and
4. the postcondition.

The name is a nonempty string. Priorities are used by each agent to decide which of the possibly several rules that apply to its situation to execute. The priority of a rule is a positive integer between 1 and 32767. Higher priorities win over lower ones. The precondition defines whether or not an agent may apply the rule. Finally, the postcondition defines the actions to be performed when a rule is applied to an agent.

2.1 Precondition

The precondition of a rule is any boolean combination of: compare priorities, check neighboring empty/filled positions, check own connections, match states/text or counters/integers, and compare calculation results with counters, messages and integers.

More precisely: a precondition is an *AND* combination of the following.

Neighbors `N____[__]`
The situation of the direct neighboring positions (north, west, east, south[, above, below]). For each of them, `0` denotes empty (no agent), `1` denotes filled (an agent), and `*` denotes indifferent.

Empty position `Edx,dy[,dz]`, `EC__,dy[,dz]`,
An empty position requirement. Written as an E, followed by the relative coordinates of the lattice position required to be empty, separated by a comma. Alternatively instead of each value `dx,dy` or `dz` the name of any counter can be inserted, where a counter starts with a `C`, followed by the two digits number of the counter.

Filled position `Fdx,dy[,dz]`
A filled position requirement. The restrictions and the syntax are the same as in the *Empty position* condition.

Priorities `P____[__]`
Compare the priority of (the applied rule/s) of the direct neighboring agents (north, west, east, south[, above, below]) with the agents' own priority. For each of them, `<` denotes that the priority of such agent needs to be (strictly) smaller, `=` denotes smaller or equal, and `*` denotes indifferent.

Smaller Priority `Ldx,dy[,dz]`
A (strictly) less priority agent requirement. The L is followed by the relative coordinates of the agent required to have smaller priority, separated by a comma. The usage of counters is the same as in the *Empty position* condition.

Smaller or equal Priority `Qdx,dy[,dz]`
A less or equal priority agent requirement. Syntax and usage is analogous to the *Smaller Priority* condition.

Attachments `A____[__]`
The attachment states to the direct neighbors (north, west, east, south[, above, below]), where `0` denotes not attached, `1` denotes attached, and `*` denotes indifferent.

State `S_____`
The agent state can be required to match a simple pattern, where an asterisk matches any character.

State of a remote agent `Tdx,dy[,dz],_____`
This is a combination of the *Filled position* and the *State* precondition. Written as a T, followed by the relative coordinates of the lattice position that needs to be filled, and ended by the state that the remote agent must have. The usage of counters is the same as in the *Empty position* condition.

(Text) messages from direct neighbors
`M*_____`, `MN_____`, `MW_____`, `ME_____`, `MS_____`, `[`, `MA_____`, `MB_____]`
Every agent has four [six] text messages from its direct neighbors (`*`=any, `N`=north, `W`=west, `E`=east, `S`=south[, `A`=above, `B`=below]), each consisting of exactly 5 characters. Any of these messages can be required to match a pattern, where an asterisk matches any character and at most four asterisks are allowed.

Numeric comparisons `<(-)____ (-)____`, `>(-)____ (-)____`, `=(-)____ (-)____`
In addition to its 25 [45] counters, every agent has $4 * 8 = 32$ [$6 * 3 = 18$] numeric messages from its direct neighbors, denoted `#N01,...,#N08`, `#W01,...,#W08`, `#E01,...,#E08`, `#S01,...,#S08` in 2D, and limited to 3 counters per direction in 3D, including `#A01,...,#A03`, and `#B01,...,#B03`. Asterisks can be used instead of a specific direction. Any of these numeric values can be required to fulfill a comparison with respect to any other such value or to any four digit number.

Remote numeric comparisons
`Vdx,dy,C___ (-)____`, `Wdx,dy,C___ (-)____`
These options are only available in 2D. They allow to compare the first value with the second value. `V` indicates strictly smaller and `W` indicates smaller or equal. The first numeric value is a counter from a remote agent at relative coordinates `dx,dy`. It requires the agent to exist. The second numeric value can be a counter, a numeric message from a neighbor or any four digit number. See more details in the *Numeric comparisons* description.

The following two operators enable generating any boolean combination:

Parenthesis `( )`
Group the expressions they surround.

Negation `!`
Negates the expression it precedes.

2.2 Postcondition

The postcondition of a rule defines the actions performed by an agent when it applies the rule. It is any *AND* combination of the following: change position, change attachments, modify state, compute and update counters, and send messages. More precisely:

Position change `Pdx,dy[,dz]`
Move the agent to the given relative coordinates. Counters can be used as in the *Empty position* condition.

Attachments `A____[__]`
For each of the four [six] possible directions (in the already described order), the possibilities are: `0` detach (if attached) before moving, and stay detached afterwards; `1` detach (if attached) before moving, and attach afterwards (if possible); `*` detach (if attached) before moving, and attach afterwards if attached before (and possible); `+` stay attached along the movement. In this case, attached agents are carried along with the moving agent.

State `S_____`
New state of the agent. An asterisk denotes that the according character remains unchanged.

(Text) messages to direct neighbors
`M*_____`, `MN_____`, `MW_____`, `ME_____`, `MS_____`, `[`, `MA_____`, `MB_____]`
Send messages to neighbors (`*` = all).

Calculations on counters and numerical messages `C___ _ ____ ____`, `#___ _ ____ ____`
2D only:
`C___ _ dx,dy,C____ ____`,
`#___ _ dx,dy,C____ ____`,
`C___ _ C__,C__,C____ ____`,
`#___ _ C__,C__,C____ ____`
Every calculation action starts with a position to write the result to (counter or outgoing message), followed by the operation to be performed, and two (readable) values on which the operation is performed. Possible operations are `+` (add), `-` (subtract), `*` (multiply), `/` (divide), `M` (modulo), `A` (maximum), and `I` (minimum). As values for an operation, either four-digit-numbers or internal counters (or one external counter, only in 2D) or incoming numerical messages can be used. The external counter is defined by first indicating the coordinates (`dx,dy` or `C__,C__`) of the agent and then the counter.

Swap `XN`, `XW`, `XE`, `XS`, `[XA`, `XB]`
Exchange the positions of two neighboring agents. Written as a `X`, followed by the desired swap neighbor.

3 The program flow

The program synchronously runs the rules on the agents. It starts by reading the initial setting as well as the set of rules. At every step, the following operations are performed in the order listed below. Alternatively, the order of steps 2 and 3 can be transposed by the user, if desired. It is also possible for the user to make all rules not involving position changes to be applied before those involving position changes.

1. Check and get valid rules. For all agents, check which rules would apply (ignoring priorities) and store valid rules sorted by priority. Store the highest priority of valid rules as current priority and set the agents priority to *open*. For all *open* priority agents sorted by priority, do until all agents have *fixed* priority: i) fetch current rules to current priority, and ii) check priority-conditions for all rules. If they are fulfilled, set priority to *fixed*. If a condition is not fulfilled, remove this rule from the specified agent and if the agents rule list is empty, reduce the current priority to the highest priority of the remaining rules. If a circular dependency between rules on different agents is detected, remove all related rules. Finally, for each agent remove all rules with priority lower than the priority of the agent.

2. Perform actions. For all agents, for all previously stored rules for the agent, perform applicable actions in the following order: i) detach, ii) compute attachment decisions, iii) change position (includes collision detection test), iv) update attachments, v) update state, and vi) swap agents.

3. Compute calculations and send messages. For all agents and for all previously stored rules for the agent, do all calculations (in the order they are listed in the rule) and send numerical messages and all text messages to the post-office. Then, deliver all messages from the post-office to their recipients.

4 The interface

The main window of the program consists of a menu and a tabbed panel with five tabs, as can be seen in the topmost portion of Figure 1.

Universe. This tab allows to visualize the agents as they apply the rules. The algorithm can be visualized

step by step or can be let to run, it can be stopped, and it is also possible to jump one or more steps forwards and backwards. In addition to the colors that can be used to distinguish the agents' states and their attachments, clicking on an agent allows to show its id, position, attachments, state, counter values, messages, and current priority. Zooming and translating the scene is always possible. In the 3-dimensional simulator rotations are also possible. Figure 1 shows a screen shot of the universe of the 2D simulator, in which the information of one of the agents can be seen. The universe panel also shows the current number of iterations, and all warning and error messages.

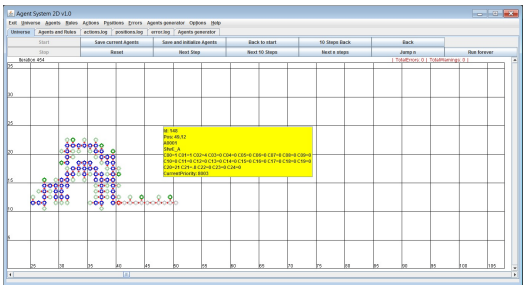

Figure 1: A screen shot of the program, showing the visualization of a set of rules running on a set of agents.

Agents and Rules. The tab consists of two text panels. The left one shows the agents file, the right one shows the rules file. Both files can be independently loaded, modified and saved. Editing shortcuts are provided. When saving any of the files, inconsistencies and syntax errors are detected and marked. See Figure 2 for an illustration.

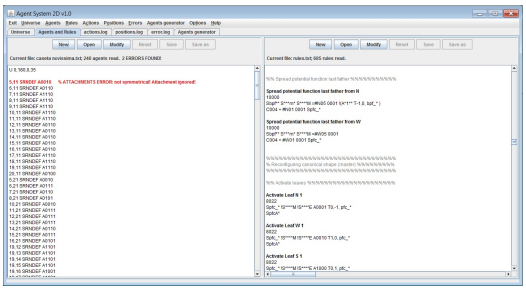

Figure 2: A screen shot of the panel showing the current agents (left) and rules (right). An error detection is shown.

Log tabs. There are three log tabs, each showing the corresponding file. The *actions.log* file stores the information of the rules applied by all agents at each iteration. The *positions.log* file stores the complete information of all agents at each iteration (position, state, attachments, counters, etc.). The *error.log* file stores all error messages at each iteration.

Agents generator. This tab allows to graphically generate or modify a set of agents, together with their attachments, states and counters.

5 Implemented algorithms

We have designed and implemented a large set of distributed algorithms, and we have run them on different configurations of agents. The implemented algorithms cover tasks from self-organization to self-reconfiguration. Self-organization includes: choosing a leader, building a spanning tree, counting the number of agents, and computing the minimum bounding box. All these self-organization tasks refer to connected sets of agents. Details can be found in [4]. Among the self-reconfiguration algorithms, we have implemented generic reconfiguration strategies for arbitrary connected shapes either assuming linear force per module [4], inspired by the centralized algorithm proposed in [1], or only constant force [3], following [2]. In addition, we have also implemented path finding algorithms, as well as some screen-saver-like amusement ones.

6 Conclusion

Our simulator is robust, and it is our strong belief that it will be useful to researchers wishing to run experiments on a wide range of distributed algorithms for self-organizing agents. For practical purposes the system scales linearly in nk, where n is the number of agents and k is the number of rules.

We therefore offer both the simulator and the aforementioned examples to the scientific community. They can be downloaded from the web page [6], which also includes i) the source files, ii) a user guide, and iii) the details of the already implemented algorithms.

References

[1] G. Aloupis, S. Collette, M. Damian, E. D. Demaine, R. Flatland, S. Langerman, J. O'Rourke, S. Ramaswami, V. Sacristán, S. Wuhrer, Linear reconfiguration of cube-style modular robots, *Computational Geometry – Theory and Applications*, **42, 6-7** (2009), 652–663.

[2] F. Hurtado, E. Molina, S. Ramaswami, V. Sacristán, Distributed universal reconfiguration of 2D lattice-based modular robots, in: *Proc. 29th European Workshop on Computational Geometry*, 2013, 139–142.

[3] O. Rodríguez, *Simulació de l'actuació distribuïda de robots modulars*, Degree thesis, Universitat Politècnica de Catalunya, Spain, 2013.

[4] R. Wallner, *A System of Autonomously Self-Reconfigurable Agents*, Degree thesis, Graz University of Technology, Austria, 2009.

[5] http://cs.gmu.edu/∼eclab/projects/mason/
Last visited: May 15, 2013.

[6] http://www-ma2.upc.edu/vera/AgentSystems/