DynaGraph : a Smalltalk Environment for Self-Reconfigurable Robots Simulation

Samir Saidani

Laboratoire GREYC, Université de Caen, France, saidani@info.unicaen.fr

Michaël Piel

Laboratoire GREYC, Université de Caen, France, mpiel@etu.info.unicaen.fr

Abstract

In the field of self-reconfigurable robots, a crucial problem is to model a modular robot transforming itself from one shape to another one. Graph Theory could be the right framework since it is widely used to model different kind of networks. However this theory in its present state is not suitable to model dynamic networks, *i.e.* networks whose topology changes over time. We propose to inject a dynamic component into graph theory, which allows us to talk about dynamic graphs in the sense of a discrete dynamical system. To address this problem, we present in this paper the theoretical framework of dynamic graphs. Then we describe the Smalltalk implementation of a dynamic graph, allowing us to perform simulations useful to understand the dynamics of such graphs.

Key words:
Self-Reconfigurable Robots, Dynamic Graphs, Dynamic Cellular Automata

1 Introduction

1.1 Self-Reconfigurable Robots

Our work is situated in the more general field of modular robotics research and specifically in the area of self-reconfigurable robots. Self-reconfigurable robots have the ability to adapt to their physical environment and their required task by changing shape. Some of the self-reconfiguring robot systems are heterogeneous : some modules are different from the others, whereas in homogeneous approach all the modules are identical.

In the literature, we distinguish two kinds of work. The first one tries to define elementary modules which can be human-assembled to rapidly build robots dealing with specific problems. For example, RMMS [1] applies this approach to build manipulators, as well as the Golem project [2] where "robots have been robotically designed and robotically fabricated" and Swarm-bots [3] for the design and implementation of self-organizing and self-assembling artifacts.

On the other hand, modular robotics look for self-reconfigurable structures. In this case, the problem is often to build identical components which dynamically reconfigure themselves to adapt their behavior to a specific task. Some works already done on this kind of self-organizing components include the Molecule Robots [4], based on a single component with two elementary movements, USC/ISI Conro [5] assembled as a serial chain with two degrees of freedom and docking/dedocking connectors, I-Cube [6] modules, quite similar but with three rotations. In the Crystalline robot [7], the elementary component uses a 2D translation movement, Telecube [8] implements a 3D version of this Crystalline component, PolyBot/Polypod [9] has a very rich structured based on simple components and finally the MEL [10] proposes a two-rotation element with a universal connecting plate allowing dynamic coupling. There is already a lot of proposals, but we think there is still an interest in building another kind of robotic atom : in our project, each module can move around autonomously and connect each other to form a new robot. In this manner, they act as ants : they can cooperate to achieve various tasks or they can form a new structure by connecting themselves to cross a hole for instance.

1.2 The MAAM Project

The MAAM [1] project, supported by the Robea project of the French CNRS, aims at specifying, designing and realizing a set of elementary robots able to connect each other to build reconfigurable systems. The self-reconfigurable robot is based on a basic component called atom. An atom is defined as a body with 6 orthogonal legs (see Figure 1). Each leg moves in a cone around his axis and the extremity of a leg holds a connector allowing connection between atom legs.

The goal of MAAM is to leverage this kind of atom to build dynamic reconfigurable structures called molecules. An advantage of such reconfigurable structures is that atoms move autonomously and they find and move to other atoms to connect with them by using their sensors on the end of each leg. In a 2D plan, they build an active carpet by array connection. A snake robot can be assembled to move into encumbered world. With a 3D structure, the molecules climb on objects, transform itself into a tool, surround objects and

[1] MAAM is a recursive acronym that means : Molecule = Atom | Atom + Molecule

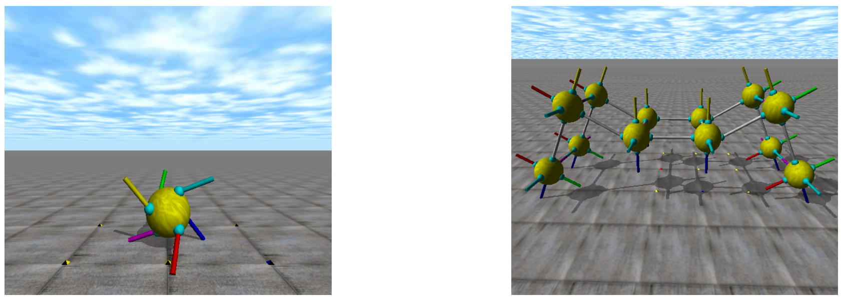

(a) An atom (b) A spider

Fig. 1. MAAM Project atoms and molecules

manipulate them. This project leveraged the cooperation of researchers from different fields : robotics, electronics, computer science, mechanics... Several french universities are involved in this project: Université de Paris VI (LIP6), Université de Caen (GREYC), Université de Bretagne Sud (LESTER and VALORIA), Laboratoire de Robotique de Paris.

1.3 Self-Reconfiguration Algorithms

A self-reconfigurable robot is ideally composed of cheap elementary units. The cheapness constraint makes an industrial manufacture possible and has a direct consequence on self-reconfiguration algorithms design. These algorithms must be the simplest one since the power calculus of the units is very weak. Therefore we think that reconfiguration algorithms requiring the description of the target shape are not relevant in this perspective.

So we are especially interested in distributed reconfiguration algorithm not requiring the exact description of the target shape. Thus Bojinov et al. (2000) [11], [12] proposed biologically inspired control algorithms for chain robots, using growth, seeds and scents concepts to make the target shape emerge from local rules. Another approach designed by Butler et al. (2003) [13] [14] is based on architecture-independent locomotion algorithms for lattice robots, inspired by the cellular automata model. Abrams and Ghrist (2003) [15] considered geometrical properties on a shape configuration space adapted to parallelization.

Actually we are looking for a general approach to tackle self-reconfiguration for different kind of modular robots. Modular robots are modules networks, and networks are usually modeled by graphs, ideal to stress the relation between entities. We can for example express the modules connectivity by bounding the degrees of a graph, unidirectional and bidirectional connections by directed or undirected edges... However, the reconfiguration of a modular robot implies the evolution of the modules network topology, and thus of its underlying graph.

Modeling the evolution of a network topology is quite hard to capture in graph theoretical model, which is essentially static. The fundamental work on random graphs, by Erdös and Rényi [16] was the very first attempt to add dynamicity in a graph. Recently, interest has grown among graph theoretician in dynamic graphs, and especially in dynamic algorithms able to incrementally update a solution on a graph while the graph changes. To represent a dynamic graph, Harary (1997) [17] proposed dynamic graph models based on logic programming and the study of the sequence of static graphs. Ferreira (2002) [18] proposed a model called "evolving graph", whose definition is based upon an ordered sequence of subgraphs of a given digraph. But this given digraph induces an *a priori* knowledge on the dynamic process.

In the following sections, we will try to answer two main questions : how to model the modules network of a self-reconfigurable robot and how to design self-reconfiguration algorithms so that a self-reconfigurable robot converges to the required shape ?

2 Modeling Self-Reconfigurable Robots with Dynamic Graphs

2.1 Emergent Calculus

We are looking here for a way to control the convergence of dynamic graphs - modeling modules network - to a target topology by emergent calculus, that is to say the modules do not know the goal configuration and the final configuration emerges from the modules collective behavior.

We would like too to found the notion of dynamic graph in the area of dynamical systems. A dynamical system is characterized by a configuration space and a function defined on this space: the goal is to understand the dynamical behavior of this function according to the structure of the configuration space and to the property of the function. Dynamical system theory is a rich and well developed field and we hope to benefit of research and results of this field by defining a dynamic graph as a special dynamical system. Moreover, extending the modeling power of graph theory on dynamic network should be interesting since dynamic networks seem to appear fundamental in very different fields as it was recently stressed by Albert and Barabasi [19].

In addition, to take into account the distributed nature of a lot of observed phenomena, we would like the graph topology to evolve in a decentralized way with local and simple rules : each node has a local knowledge on its neighborhood and a limited power for computation and communication. Cellular automata are well known for their ability to express complex dynamics from the local knowledge of the cells. The underlying lattice of a cellular automata is usually static, but Ilachinski and Halpern (1997) [20] developed a cellular automata model in which the underlying infinite d-dimensional array (*i.e.* the metric space $\mathbb{Z}^d$) evolves according to link transition rules. But the very formulation of this model, expressed through the array data structure, is not relevant to express a graph topology. Let us note also that link transition rules depend on the states of cells neighborhood and we are looking for purely "topological" rules.

To combine graph theory expressness with richness of cellular automata dynamic, we define the notion of topodynamic of a graph, with the assumption that a module only knows about its neighborhood and the neighbors of its neighborhood.

The last part is devoted to the validation of the framework proposed in the following section, by building a dynamic graph implementation in Squeak [21]: we present the overall architecture behind the DynaGraph software and show an example of simulation where the final topology emerges from the dynamic nodes collective behavior.

By this way, we hope to transform the shape controlling problem into the study of graph topodynamic, where simulation should be a valuable tool in discovering of topodynamics converging towards a given topology.

2.2 Topodynamic of a Graph

We first remind basic notions in graph theory and then state a topological definition of a graph independent of its embedment in metric space. We finally define the notion of graph topodynamic.

2.2.1 Preliminaries

Definition 2.1 (Graph) *A graph is a pair (V, E) with V a finite set of vertices and E a set of edges, finite subset of $V \times V$.*

A graph is undirected if the relation defined on V is symmetric, otherwise the graph is directed, and edges have a direction. The *order* of a graph is its number of vertices $|V|$. Two vertices are said to be *adjacent* if they are joined by an edge.

The *neighborhood* of a vertex v is the set $\tau(v)$ of vertices such that they are adjacent to v. If there is no ambiguity with the context, we note a neighborhood $\tau(v) = \{x, y, \dots, z\}$ by $xy \dots z$. The *out-neighborhood* of a vertex v is the set $\tau^+(v)$ of vertices outgoing from v. The *in-neighborhood* of a vertex v is the set $\tau^-(v)$ of vertices pointing to v. We see that an undirected graph (resp. directed graph) is completely described by giving the set of its vertices and a neighborhood (resp. in-neighborhood *or* out-neighborhood) on each vertex. The *degree* (resp. *indegree*, *outdegree*) of a vertex is the number of its neighbors (resp. in-neighbors, out-neighbors). The degree of a graph, noted $d^\circ(G)$ is the maximum degree of a vertex. Note that for an undirected graph, each edge v, w could be considered as a double arrow (v, w) and (w, v), so the in-neighborhood is equal to the out-neighborhood of a vertex.

Definition 2.2 (Graph Topology) *The topology τ (resp. τ^+, τ^-) of a graph G is the family of its neighborhoods $(\tau_v)_{v\in V}$ (resp. out-neighborhoods $(\tau_v^+)_{v\in V}$, in-neighborhoods $(\tau_v^-)_{v\in V}$).*

Example 2.3 *Let G be the following graph :*

$$\begin{tikzcd}
a \arrow[d, leftrightarrow] \arrow[dr] & & \\
b & c \arrow[d] & \\
 & d \arrow[r, leftrightarrow] & e \arrow[ul]
\end{tikzcd}$$

The neighborhood $\tau(a)$ of a is $\{b, c\}$ or in short bc. We have also the following relations: $\tau^+(a) = bc$, $\tau^-(a) = b$, $\tau(c) = ade$, $\tau^+(c) = d$, $\tau^-(c) = ae$, $d^\circ(G) = 3$ because $|\tau(c)| = 3$. The graph topology is the family $(\tau^+(a), \tau^+(b), \tau^+(c), \tau^+(d), \tau^+(e))$

2.2.2 Topodynamic

Let us now define the notion of a sequence of graphs.

Definition 2.4 (Sequence of Graphs) *A sequence of graphs is a family of graph $(\mathcal{G}_i)_{i\in\mathbb{N}}$ with $\mathcal{G}_i = (V_i, \tau_i)$.*

For simplicity, we consider from now only sequence of graphs with constant order, *i.e.* $\forall i \in \mathbb{N}$, $V_i = V_0$.

What is the difference between a sequence of graphs and a dynamic graph ? Usually, a dynamic system is characterized by its transition function : we can compute the state of the system from an initial state and past states.

Basically, the graphs in a sequence of graphs cannot change their topology on their own : the evolution of the topology is predetermined by giving a family τ_i of topologies. However, we can associate a transformation function to a sequence of graphs. So we call dynamic graph a sequence of graphs consisting of an initial graph and a function which transforms its topology to a new topology.

Definition 2.5 (dynamic graph - global transition function) *A dynamic graph is the pair* (G_0, Δ)*, such that* $G_0 = (V, \tau_0)$ *is an initial graph and* $\Delta : (V \mapsto 2^V) \mapsto (V \mapsto 2^V)$ *define a* topodynamic *by mapping a topology on* V *to a new topology.*

So the topodynamic is an algorithm taking as input a graph and giving as output a graph of the same order. More precisely, this algorithm takes a node and its neighborhood of the input graph and computes a new neighborhood for this node. It continues to compute over all the nodes of the input graph to render the output graph. Then this output graph becomes the input graph and we apply the topodynamic again. This whole process draws the dynamic graph.

Nevertheless, we would like to have dynamic graph vertices more active than in a sequence of graphs, namely able to change their own degrees by accepting, keeping or removing its adjacent edges, according to local transition rules inducing the graph topodynamic. Local transition rules are widely used in cellular automata area: the state of an automaton depends on its own state and the state of its neighbors. Local transition function, simultaneously applied to each cell, determine the dynamic of a cellular automaton. Although cellular automata are usually defined on regular lattices, this definition can be extended to more complicated graph: graph of automata (connected bounded degree graph), first introduced by Rosenstiehl (1966) [22]. In a graph of automata, each node has a state and the next state depends of its current state and the state of its neighbors.

Definition 2.6 (graph of automata) *A graph of automata is a triplet* (S, G, δ) *where* S *is a finite set called set of states,* $G = (V, \tau)$ *is a graph,* $\delta : S \times \{(S \cup \epsilon)^{d^\circ(G)} \diagup \sigma\} \mapsto S$ *is the transition function where* ϵ *is a special element used when the vertex has less than the maximum degree of the graph.* σ *is the equivalence relation defined on the cartesian product* S^n *with* $x \sigma y$ *if* x *is a permutation of* y*. So* $S^n \diagup \sigma$ *is the unordered set* S^n*.*

In this definition of graph automata , the underlying graph is static. We study here the possibility to have an evolving underlying graph : this evolution may be controlled by active vertices, kind of automata able to connect and disconnect their own edges in the network. We give to the automata the control of its underlying graph by slightly modifying the graph automata definition as following.

Definition 2.7 (dynamic graph - local transition function) *A dynamic graph is the pair (G_0, δ) where $G = (V, \tau_0)$ define an initial graph, and $\delta : S \times \{(S \cup \epsilon)^{|V|-1} / \sigma\} \mapsto S$ with $S = 2^V$ the set of states, define the local transition function, where ϵ is a special element used when the vertex has less than $|V| - 1$ (the maximal degree of the dynamic graph).*

A node chooses its next neighborhood according to its current neighborhood and the current neighborhood of its neighbors. If we replace "neighborhood" in the precedent sentence by the word "state", we retrieve the usual definition of the evolution of a cell in cellular automata.

To deal with the different neighbors of a given neighborhood, we build from the application τ an application $\vec{\tau}$ which for each vertex gives its *neighbors vector* : $\vec{\tau} : \{v\} \mapsto (\{1, \ldots, |\tau(v)|\} \mapsto V)$ such that $\bigcup_{i=1}^{|\tau(v)|} \vec{\tau}(v)(i) = \tau(v)$ where $v \in V$

If a vertex v has a degree n, its next state $\tau_{i+1}(v)$, namely its next neighborhood, is given by :

$$\tau_{i+1}(v) = \delta(\tau_i(v), \tau_i(\vec{\tau_i}(v)(1)), \ldots, \tau_i(\vec{\tau}(v)(n)), \epsilon, \ldots, \epsilon)$$

We define now the fix-point topology for a given topodynamic as a graph topology unchanged by applying this topodynamic.

Definition 2.8 (fix-point topology) *A fix-point topology τ for the topodynamic Δ is a topology such that $\Delta(\tau) = \tau$*

A question we may ask is for which topodynamic a given topology is the fix-point, *i.e.* the so-called *inverse dynamic problem.* For the moment the design of such topodynamic rules is rather a matter of intuition supported by computer experimentations, but resolving the *inverse dynamic problem* would provide us with a means of constructing a desired topology, or at least to show us the impossibility to automate this construction. Anyway the simulation of dynamic graphs should be valuable to address this problem.

3 DynaGraph : an Environment for Dynamic Graph Simulation

We present here the DynaGraph environment intended to understand and simulate the dynamic of a graph, following the theoretical framework stated in the first section. Through this environment, we hope a better understanding of the dynamic graphs behavior. For instance, we could ask what kind of dynamic graph converges to a path graph when the initial graph is a star graph, so we have to define the relevant rules allowing such a convergence. We also need to implement this rules to gain trust on this rules before proving their correctness.

3.1 The DynaGraph Environment

Let us imagine that we want to reconfigure a spider robot, represented to the left of Figure 2, to a caterpillar robot. Each node represents a module, and directed edges the connection between modules.

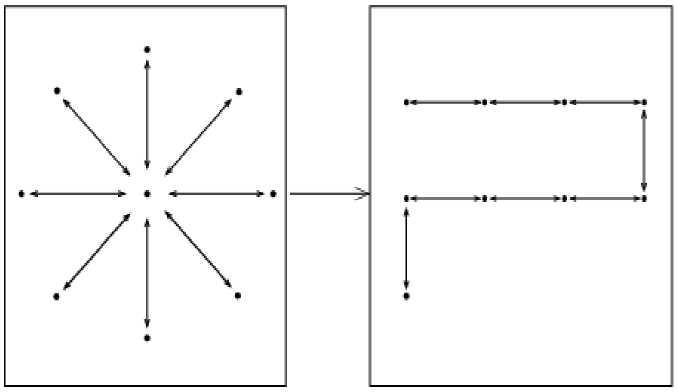


Fig. 2. A star to chain reconfiguration

The expression `DynaGraphSystemWindow open` launches the DynaGraph GUI (Fig. 3).

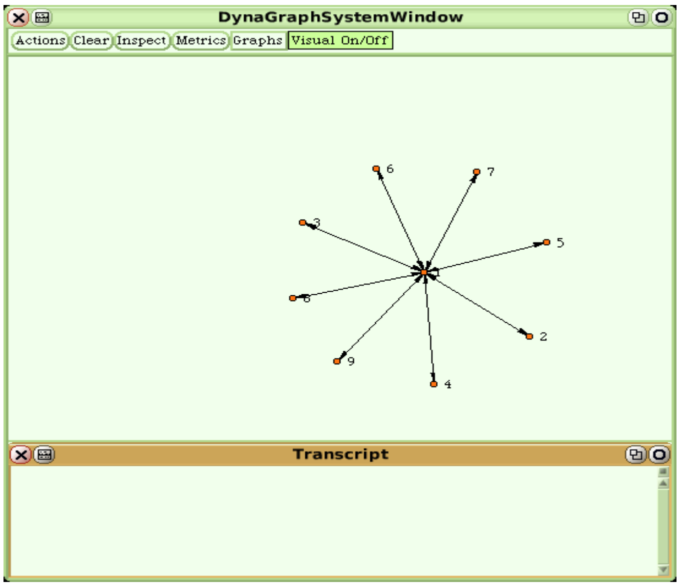


Fig. 3. The DynaGraph System Window

The DynaGraph window (Fig. 3) is currently composed of two parts : the first one is the gui interface and the second one is for debugging purpose.

Several buttons allow the user to start, stop or run the simulation step by step. The user chooses the initial graph of the dynamic graph thanks to the `Graphs` button. Several metrics *i.e.* function of the graph at the current step are available, like degree distribution, clustering coefficient and average path length, to gain some information during the evolution of the dynamic graph. We use the `PlotMorph` package [23] to display such kind of informations (see Figure 4).

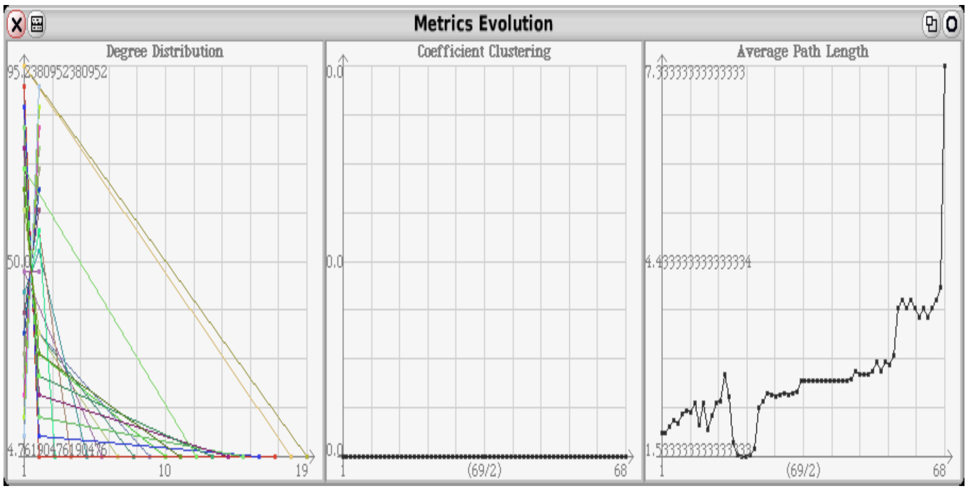


Fig. 4. Dynagraph metrics

We have enhanced this package to export the curves in the gnuplot format to exploit thouroughly the collected data. Note that we have also implemented a GraphSystemWindow to deal with graphs, *e.g.* random graphs, so the metrics are available for this graphs. For instance, Figure 5 shows the degree distribution of a single random graph of 10000 nodes and a connection probability of 0.0015, generated through the `exampleRandomGraphWithOrder:probability:` method. We verify that the deviation is small between the generated random graph and the theoretical result stating that for large N, the probability for a node to have a degree k follows roughly a Poisson distribution $P(k) \simeq e^{-pN \frac{(pN)^k}{k!}}$.

The simulation shows in Figure 6 that after few steps the initial star topology is transformed into an undirected acyclic graph to finally reach a fix-point. The local topodynamic rules, when applied to the neighborhood of each node, does not change their neighborhood anymore. We show thanks this simulation that it is possible to make a topology emerge from a local topodynamic rule.

3.2 Overview of the DynaGraph Architecture

The dynamic graph classes are build on top of graph classes, designed in Smalltalk by Mario Wolczko and ported for Squeak by Gerardo Richarte and

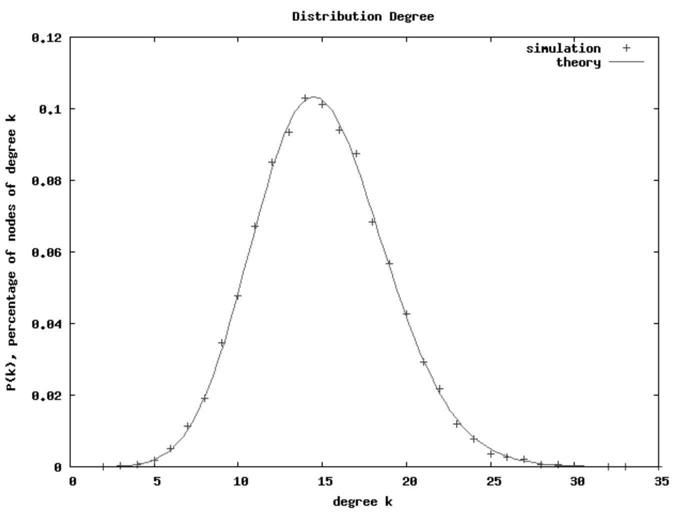
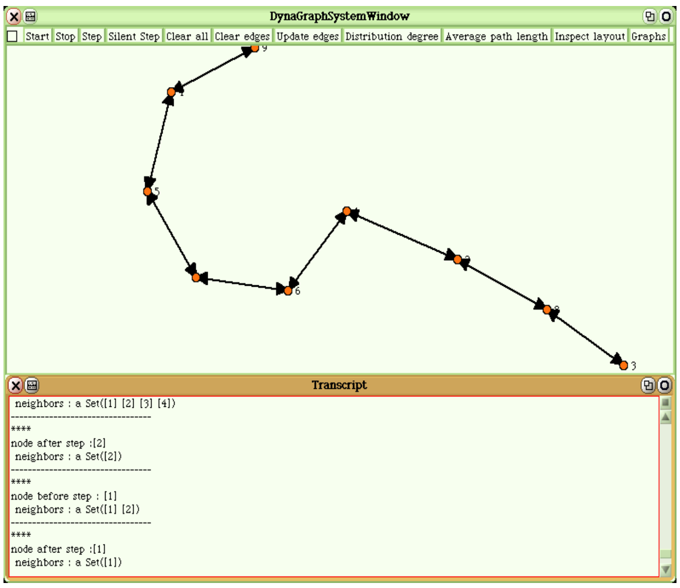


Fig. 5. The degree distribution resulting from the simulation of a random graph ($N = 10,000$ nodes, connection probability $p = 0.0015$).



```
 neighbors : a Set([1] [2] [3] [4])
--------------------------------
****
node after step :[2]
 neighbors : a Set([2])
--------------------------------
****
node before step : [1]
 neighbors : a Set([1] [2])
--------------------------------
****
node after step :[1]
 neighbors : a Set([1])
```

Fig. 6. The chain, fixpoint topology

Luciano Notarfrancesco. A GraphMorph class is available, allowing us to layout a graph through two main methods : a springs-gravity model where each node behave like negative electrical particles and an animated radial layout. We use the springs-gravity model to display a dynamic graph.

Several principles are used behind a local topodynamic design. These principles are not really specific to one particular transformation, but they are rather a guide for designing a topodynamic :

- Reconfiguration : the reconfiguration is done thanks to the knowledge of a neighborhood node and of the neighborhood of the node neighbors.

It can disconnect from its current nodes to reconnect itself to a neighbor of its neighbors.

- Local knowledge : A node knows only its own indegree and outdegree (computed from its knowledge of its in and out-neighbors) and the in and outdegrees of its direct neighbors (computed from its knowledge of in and out-neighbors of its neighbors).
- Outgoing connection control : A node only controls its outgoing connections, it cannot decide to disconnect itself from an ingoing connection but can connect to or disconnect from its outgoing connections.
- Decision process : To take a decision, for instance a reconnection to another node or a disconnection, a node may exploit the dissymmetry of its neighborhood. Indeed if a node wants to reconnect itself to a neighbor of one of its neighbors, it has to check its neighbors degree and take a decision according to the neighbors degree. If all neighbors have the same degree, the reconnection to the neighborhood neighbors will be random : from the node point of view, there is no way to decide which neighbor of its neighbors to choose since all its neighbors have the same degree. Otherwise it will exploit the dissymmetry related to the difference of degree. For that reason, we have to avoid the ring topology because of the possibility to lose graph connectivity.
- Connectivity : A node must never be isolated during the reconfiguration process.
- Uniformity : All nodes have the same set of rules.
- Synchronicity : The topodynamic rules, *i.e.* a combination of disconnection and connection rules, are applied *simultaneously* over all the nodes. The computation must hold synchronous nodes reconfiguration although it could be interesting to study how asynchronicity influences the reconfiguration process. But we choose to study first a synchronous dynamic since it is the simplest one : we have not to choose which nodes the rules must apply first, all nodes are equivalent according to the application of rules.

These principles constraint the implementation of a dynamic graph : for instance, synchronicity means that we have to take care about the way each node changes its neighborhood. Indeed if a node changes its neighborhood, one of the neighbor of this node must not see immediately this change... The global architecture of a dynamic graph, or dynagraph, is pictured in Figure 7.

Moreover, to control the evolution of a dynamic graph, we have added the `MetaGraph` class, which is composed of the list of graphs computed during the evolution of a dynamic graph. This class could be seen as an equivalent of the space-time diagram used in cellular automata simulation and allows us to keep trace of the different states of a dynamic graph. The more important and critical method of this class is the method `oneStep` which is implemented as following :

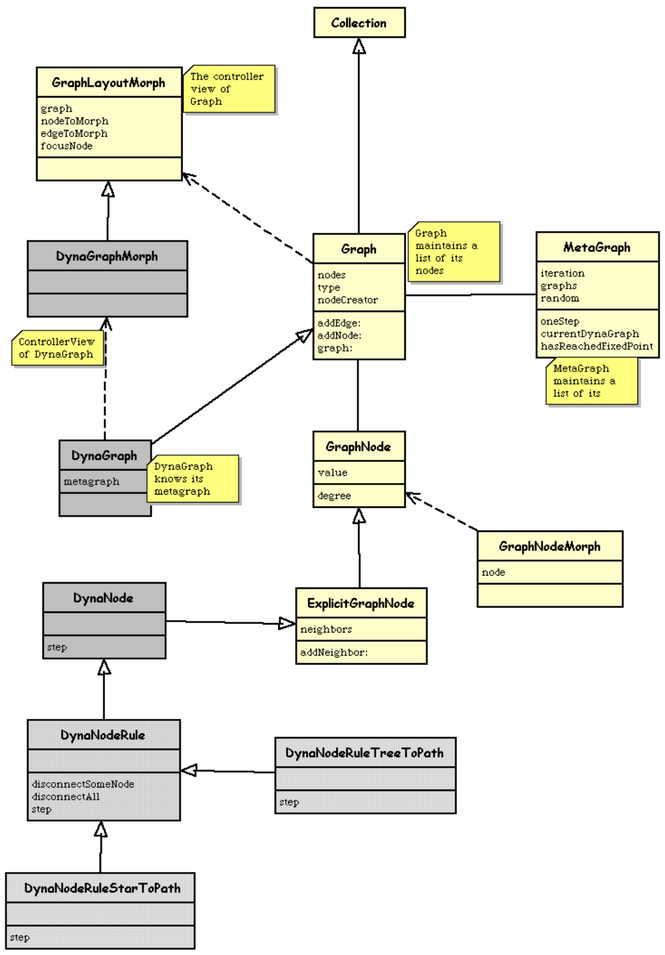


Fig. 7. The DynaGraph Architecture

```
| nextDynaGraph |

nextDynaGraph := self currentDynaGraph veryDeepCopy.
"we enter to the next iteration, all nodes will refer
to the precedent iteration to compute their neighborhood"
iteration := iteration + 1.
(nextDynaGraph nodes asArray shuffledBy: random)
        do: [:node | node step].
self addGraph: nextDynaGraph.
^ nextDynaGraph
```

Let us take an example to understand how to get synchronous processes. Suppose that there is only one rule applied to all nodes : a node disconnects from the other one if and only if its indegree is 1, that is to say if there is a node connected to itself.

$$\begin{tikzcd}\bullet_1 \arrow[r, leftrightarrow] & \bullet_2\end{tikzcd}$$

The node labelled 1 is in this case, so it will disconnect from the node 2 :

$$\begin{tikzcd}\bullet_1 & \bullet_2 \arrow[l]\end{tikzcd}$$

If all this process was sequential, at the next step, the node 2 will keep its connection since its indegree is now O, but we would like to focus our study on parallel updates. So this behavior is not the desired one, the right result for parallel update is the following:

$$\begin{tikzcd}\bullet_1 & \bullet_2\end{tikzcd}$$

So we will copy the current graph to a new graph, then we will modify the nodes neighbors of the new graph according to the neighborhood, and neighbors neighborhood of the previous graph. In this manner, we simulate the fact that each node changes their neighborhood simultaneously.

3.3 Overview of the Implementation

The `DynaNode` class corresponds to the notion of dynamic node as explained in the Topodynamic section and implements the abstract method `step`. A dynamic node is essentially a set of rules defining the dynamic aspect of a node, so it makes sense to derive the `DynaNodeRule` class from the `DynaNode` class. Then we can use this facility to give a name to a specific `DynaNode` like `DynaNodeRuleStarToPath` and so on. The `DynaNodeRule` class implements the main methods allowing the evolution of the neighborhood of a given graph. Here is a sample of main methods involved in the graph reconfiguration:

- The method `disconnectNodeOfOutDegree: anInteger` tries to disconnect the current node from a neighbor of outdegree *anInteger* and return true. If it fails, return false.
- The method `connectToNgbOfMyNgb` connect the current node to any neighbor of its neighbor.
- The method `connectToNgbOfNgb:` connect the current node to any neighbor of its neighbor v.
- The method `connectToNodeOfInDegree: anInteger` connect the current node to a node of indegree *anInteger*.
- The method `connectToOneOfMyStrictInNgb` connect the current node to one of its strict in-neighbors, namely in-neighbors which are not out-neighbors.

- The method `connectToOutNgbOfNgb:` connect the current node to somekindof out-neighbor of its neighbor v.
- The method `connectToNode:` connect the current node to the node w.
- The method `disconnectToAnyOutNode` disconnect the current node from one out-neighbor.

From the `DynaNodeRule` class, we derive a class which implements the method `step` expressing the local topodynamic as explained in the first section, *i.e.* a topodynamic depending of the neighborhood and of the neighbors neighborhood. For instance, the method `step` of the `DynaNodeRuleStarToPath` class implements four local reconfiguration rules [24] :

```
DynaNodeRuleStarToPath>>step

        self extremeNodeClosure.
        self extremeNodeReconfiguration.
        self starLosingLeaves.
        self middleNodeReconfiguration.
        ^ self.
```

The self-reconfiguration algorithm is based on three main rules applied one time for each step :

- Reconfiguration: Threads want to find an extremity. (`extremeNodeReconfiguration` and `middleNodeReconfiguration`)
- Node closure: Closing all links when a path is formed. (`extremeNodeClosure`)
- Star losing leaves: Disconnect from some of many out-neighbors. (`starLosingLeaves`)

It would be very long to explain and develop the code of all this rules, which are currently available in the SqueakSource site under the name DynaGraph. Here is a piece of code giving an idea of how the rules are generally coded :

```
DynaNodeRuleStarToPath>>starLosingLeaves

        self currentOutDegree >= 3
                ifTrue: [self disconnectAnyOutNode]

DynaNodeRuleStarToPath>>middleNodeReconfiguration

 "principle : a middle-initial vertex walks around the graph
  till it finds an extremity"
 self isMiddleInitial
```

```
  ifTrue:
  [self currentUnclosedOutNeighbors loneElement isAnExtremity not
   & self currentUnclosedOutNeighbors loneElement isMiddle not
   ifTrue: [
    self reconnectToOutNeighborhoodOfNode:
     self unclosedOutNeighbors loneElement]]
```

Let us notice an important point when we design rules : the difference between the methods inNeighbors, outNeighbors... and the methods prefixed with the term "current", like currentInNeighbors, currentOutNeighbors... The first ones are the accessors of the nextDynaGraph (see the method oneStep above) whereas the second ones refer to the DynaGraph of the current iteration. This distinction is necessary to keep the dynamic synchronous, as it was explained above. So the conditional part of a rule must refer to the DynaGraph of the current iteration whereas the action part must refer to the DynaGraph under construction, that is to say the nextDynaGraph. So we have always to care about which kind of DynaGraph is concerned : the one of current iteration or the one in progress.

3.4 Building its own dynamic graph

To create a dynamic graph with a specific rule, we have implemented the class method `dynamicDirectedWith:` which takes as argument a DynaNodeRule class:

```
     DynaGraph dynamicDirectedWith: DynaNodeRuleStarToPath.
```

The usual way to add a new dynamic graph in the DynaGraph environment is to add a class method to the class `DynaGraph` :

```
exampleMyDynaGraph
        | d |
        d := self dynamicDirectedWith: DynaNodeMyOwnRule.
        d addEdge: 1 -> 2.
        d addEdge: 2 -> 4.
        d addEdge: 4 -> 2.
        d addEdge: 2 -> 3.
        d addEdge: 3 -> 2.
        ^ d
```

Of course, we have to first create the `DynaNodeMyOwnRule` class by deriving it from the `DynaNodeRule` class and using the methods defined in this class, or creating our own methods. The when we start the GUI, the `exampleMyDynaGraph` appears when we press the `Graphs` button in a pop-up menu :

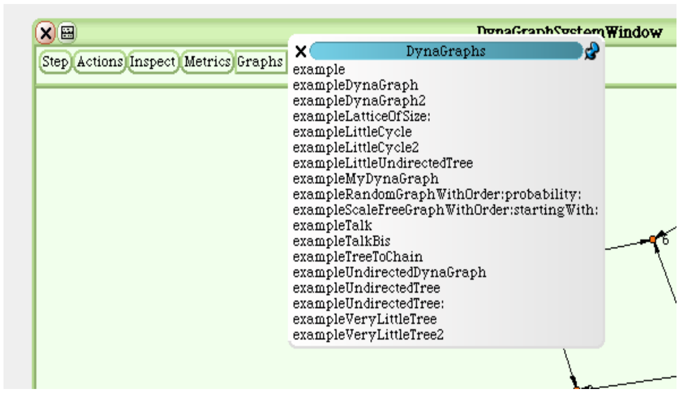


Fig. 8. Running a dynamic graph

If we want to keep trace of the dynamic graph evolution, we can use the `MetaGraph` class as following: `MetaGraph new setInitialDynaGraphTo: (DynaGraph dynamicDirectedWith: DynaNodeRuleStarToPath)`

4 Conclusion

This work presents a framework based on graph topodynamic and cellular automata intended to address the problem of controlling modules network topodynamic.

After introducing the field of self-reconfigurable robots, we have defined the notion of dynamic graph by proposing to make a distinction between sequence of graphs and dynamic graphs: a dynamic graph is a sequence of graphs where a local or global topodynamic determines the topological evolution of a graph.

Then we have described the smalltalk implementation of this framework : since a topodynamic can be defined from the local knowledge of each vertex, we can use an oriented-object language, and Squeak helps us to prototype and validate quickly and easily our framework.

The most interesting feature offered by Squeak is the debugging facilities which are crucial for developing self-reconfigurable algorithms. For example, we have implemented the inspection of a node during the simulation by shift-clicking on a node in the DynaGraphWindow. This is a great advantage compared since it is possible to modify topodynamic rules during the simulation of a dynamic graph. This possibility allows us to find out several reconfiguration algorithms in few weeks, including the time spent to develop the gui. For a research project, it is really valuable since we would like to test quickly some concepts without spending a lot of time to implement those ideas, and Squeak plays the perfect role of “notebook for concepts implementation”. But we feel at the present state of this work one limitation : it is hard to simulate dynamic graphs with a large number of nodes. The current limitation is around 30 nodes which is sufficient to test our ideas but not sufficient to gain some statistical information about the dynamic of a graph. Maybe have we now to implement some parts in C which fortunately is one of a feature of Squeak, although we would prefer to stay in the Squeak environment.

This simulator should help us in rules discovery involved in the emergence of different network topologies : from a connected graph to a chain graph, from a lattice to a chain, from a chain to a lattice, and so on. This software [25] is available as a package in the Squeakmap site, a server providing applications designed for Squeak, and is available too in the SqueakSource site.

Let us note that the MAAM project is in fact a long-term project composed of several subprojects like sQode, a plugin to interface Squeak with ODE (Open Dynamic Engine), SqueakSimulAtom, a 3D robots simulator, LCSTalk, a Learning Classifier System framework... We expect to have the first molecule with 10 elementary components in 3 years.

Acknowledgment

This work was supported by an MENRT Research Studentship, and is a part of the MAAM Project. We thank Serge Stinckwich for its valuable comments and to allow us to participate in the MAAM project, with the support of François Bourdon. We wish to thank also Stéphane Ducasse and the anonymous reviewers whose comments permit us to really improve this paper.

References

[1] C. J. J. Paredis, P. K. Khosla, Fault tolerant task execution through global trajectory planning, Reliability Engineering and System Safety (special issue on Safety of Robotic Systems) 53 (3) (1996) 225–235.

[2] H. Lipson, J. B. Pollack, Automatic design and manufacture of robotic lifeforms, Nature 406 (2000) 974–978.

[3] Swarmbots.
URL `http://www.swarmbots.org`

[4] K. Kotay, D. Rus, M. Vona, C. McGray, The self-reconfiguring molecule: Design and control algorithms, in: Workshop on Algorithmic Foundations of Robotics, 1999.

[5] W. Shen, P. Will, Docking in self-reconfigurable robots, in: Proceedings IEEE/RSJ, IROS conference, Maui, Hawaii, USA, 2001, pp. 1049–1054.

[6] C. Unsal, P. Khosla, A multi-layered planner for self-reconfiguration of a uniform group of i-cube modules, in: IROS 2001, 2001.

[7] R.Fitch, D. Rus, M.Vona, A basis for self-repair using crystalline modules, in: Proceedings of Intelligent Autonomous Systems, 2000.

[8] S. B. H. John W. Suh, M. Yim, Telecubes: Mechanical design of a module for self-reconfigurable robotics, in: Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA'02), Washington, DC, USA, 2002, pp. 4095–4101.

[9] M. Yim, D. Duff, K. Roufas, Polybot: a modular reconfigurable robot, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'00), Vol. 2, San Francisco, California, USA, 2000, pp. 1734 –1741.

[10] A. Kaminura, al, Self reconfigurable modular robot, in: Proceedings IEEE/RSJ, IROS conference, Maui, Hawaii, USA, 2001, pp. 606–612.

[11] H. Bojinov, A. Casal, T. Hogg, Multiagent control of self-reconfigurable robots Comment: 15 pages, 10 color figures, including low-resolution photos of prototype hardware.
URL `http://arXiv.org/abs/cs/0006030`

[12] H. Bojinov, A. Casal, T. Hogg, Emergent structures in modular self-reconfigurable robots, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'00), Vol. 2, San Francisco, California, USA, 2000, pp. 1734 –1741.

[13] Z. Butler, D. Rus, Distributed planning and control for modular robots with unit-compressible modules, International Journal of Robotics Research 22 (9) (2003) 699–716.

[14] Z. Butler, K. Kotay, D. Rus, K. Tomita, Generic decentralized control for a class of self-reconfigurable robots, in: Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA'02), Washington, DC, USA, 2002, pp. 809–815.

[15] A. Abrams, R. Ghrist, State complexes for metamorphic robots, International Journal of Robotics Research In press.

[16] P. Erdős, A. Rényi, On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci. 5 (1960) 17–61, a seminal paper on random graphs. Reprinted in *Paul Erdős: The Art of Counting. Selected Writings*, J.H. Spencer, Ed., Vol. 5 of the series *Mathematicians of Our Time*, MIT Press, 1973, pp. 574–617.

[17] F. Harary, G. Gupta, Dynamic graph models, Math. Comput. Modelling 25 (7) (1997) 79–87.

[18] A. Ferreira, On models and algorithms for dynamic communication networks: The case for evolving graphs, in: 4^e rencontres francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'2002), Mèze, France, 2002.

[19] R. Albert, A.-L. Barabàsi, Statistical mechanics of complex networks, Rev. Mod. Phys. 74 (2002) 47–97.

[20] Ilachinski, Halpern, Structurally dynamic cellular automata, COMPSYSTS: Complex Systems 1.

[21] Squeak, an open-source smalltalk.
URL `http://www.squeak.org`

[22] P. Rosenstiehl, Existence d'automates finis capables de s'accorder bien qu'arbitrairement connectés et nombreux, International Computer Science Bulletin 5 (1966) 245–261.

[23] Plotmorph, morphs to draw xy plots.
URL `http://minnow.cc.gatech.edu/squeak/DiegoGomezDeck`

[24] S. Saidani, Self-reconfigurable robots topodynamic, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'04), New Orleans, Louisiana, USA, 2004, pp. 2883–2887.

[25] Dynagraph, a dynamic graph simulator.
URL `http://www.squeaksource.com/DynaGraph`