RAPPORT DE RECHERCHE

L R I

SELF DEVELOPING NETWORK 1 : THE FORMAL MODEL

GRUAU F

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

01/2012

Rapport de Recherche N° 1549

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

Self Developing Network 1 : the formal model

Frederic Gruau

Laboratoire de Recherche en Informatique
Bât 650 Université Paris-Sud 11 91405 Orsay Cedex France
frederic.gruau@lri.fr `http://blob.lri.fr`

Résumé Ce travail introduit les réseaux auto-développants d'agent, pour modéliser le parallélisme existant dans le développement cellulaire biologique. Des agents à état fini exécutent des règle de réécriture locales qui créent d'autres agents et connections. Un agent initial peut ainsi progressivement développer un réseau d'agents arbitrairement grand. Nous étudions comment définir un modèle formel générique permettant une exécution distribuée, et décentralisée. Il faut imposer que lorsque un agent peut se réécrire, ses voisins ne le peuvent pas, et constituent ainsi des points d'ancrage stable. Cette première définition présente cependant un gros désavantage : c'est le programmeur qui doit assurer que son système de règle vérifie cette contrainte d'exclusion mutuelle entre voisins. Nous proposons donc une deuxième définition permettant de relaxer cette contrainte en faisant intervenir une simulation distribuée. Cette définition permet d'écrire les règles de réécritures de façon plus concise, car de plus haut niveau. Elle englobe également les modèles déjà existant sur l'auto-développement.

This work introduces Self Developing Network of agents to model the parallelism present in cellular biological development. Finite-state agents execute a local graph rewriting rule that create other agents and links. An initial ancestor agent can thus progressively develop a network of agents. We investigate how to define a formal generic model enabling a decentralized and distributed execution. One should make sure that when an agent can rewrite, its neighbors cannot, and can thus be used as stable gluing points. This first definition has a major drawback : it is the programmer task to prove that his rule system checks this constraint of mutual neighbor exclusion. We therefore put forward a second definition that can relax this constraint by means of a distributed simulation. This definition allows to write more concise rewriting rules, because of a higher level of abstraction. It also include existing models of self-development.

Keywords: Self developing network, massive parallelism

1 Motivation and review.

The cells of a biological system have an intrinsic parallelism because they exist in space and time. They can update in parallel with each other, and constantly through time. The parallelism is available *everywhere, every time.*

Moreover, cells can divide and create other cells thus increasing the available parallelism. We believe that the central property at the source of this parallelism is the ability of the system to *self develop*. It is not an external entity that builds a biological system. Instead, cells themselves carry a program, and duplicate according to it, in order to create other cells and exchange messages, which in turns create other cells, and so on, until a whole organism is unfolded. The goal of this work is to put forward a new formal model called "Self Developing Network" (SDN) that captures a similar *pan-parallel* property. An SDN configuration consists of a network of finite state agents that update in parallel in discrete time, and develop by producing other agents and connections. SDNs not only specifies a network operating in parallel, they also program a parallel development of that network. For example a 2D lattice of $2^n \times 2^n$ nodes can be developed by a process of iterative duplication, in only $2n$ steps, where each step alternatively doubles the number of rows or columns.

Self development imposes to abandon the use of a global memory. This is difficult, because programing with a BPBG [1] memory is very convenient : data is stuffed in it without worrying where, and later it is addressed whenever it is needed. Without global memory, each word of data that is created, has to know in advance *where* and *how* it is going to be used. SDNs consider active data : each data is stored as a register of an agent that has connections to other agents. The connections encode *where* the data will be used by directly pointing to the potential consumers. The agent's state encodes *how* the data will be used. Creation and deletion of data translates into creation and deletion of agents, and connections. As for biological system, the resulting development is a *self development* because the agents themselves are the actors.

Initially, the network has a minimal size, including an ancestor holding the development program, and some fixed agents used for input and output. The program let the ancestor generate new agents who also generate new agents, and so on, until a functional circuit is developed, whose structure should reflect the structure of the target problem. Alternatively, computation and computation can be interleaved with development. Many formal models consider self creation of computing elements.

Process algebra Dynamic creation of agents is a primitive operator for formal model of process algebra, which are useful to prove theorem about distributed algorithm, the Actor model [7] and [1] CSP [8], CCS [11]. PI-calculus [12]. Brane calculi [3] is used for simulating a single biological cell, where structure is light : within a cell sub compartment, any molecules can interact with any other. These models use a *global name space* because it is more simple and natural. A process can communicate with any other, using its name or id, this implicitly requires a shared memory for communication, which is not conducive to scalable parallel computing. In our view of self development, the network used for messages communication is explicitly instantiated at run time. If two agent needs to communicate, a connection must be present. Each agent has to explicitly

1. Brave Passive Big Global

manage a bounded number of connections carrying labels. It communicates by modifying those labels.

Pointer machines. In the early 1950s, Kolmogorov and Uspensky[6] outlined a concept of abstract machines that does use an explicit network representation : A configuration is a graph, instructions can add or delete new nodes, and the degree is bounded. This machine was designed with the same goal as Turing machines : to define the concept of computation. Surprisingly, the way in which the architecture is modified is not parallel : a single node is active at each time step. The graph is used as a simple storage structure more generic than the one dimensional Turing tape. This early work gave rise to several different machines called "pointer machines", reviewed in [2]. In particular, the Parallel Pointer Machine (PPM) introduced by Cook and Dymond [4] does use parallel development : The PPM consists in a collection of FSA having a fixed size array of pointers to other FSA which determine an explicit architecture. An FSA a_1 can execute an instruction to create and initialize a new FSA, or to read the state or copy the pointers from another FSA a_2 it connects with. In our view, agents should be able to update without accessing the state of neighbors, because this is one of the key for a distributed implementation.

2 Node Graph Rewriting Systems (Node-GRS)

Self Developing Networks (SDNs) consider networks whose underlying graph can be modified, the number of nodes and the set of edges can both evolve. The formal tools to modify a graph is known and studied as Graph Rewriting rule Systems [14] (GRS). A configuration is a labeled graph (node and edge). The classic form of a graph rewriting rule includes a left member and a right member which are both graphs. A rule application involves two phases : 1- Matching the left member with a sub graph of the graph being rewritten, and 2- Removing this sub-graph, adding the right member, and gluing it to the rest of the network.

SDNs can be informally defined as distributed GRS which can be executed in a decentralized distributed way. Decentralized rule application is not easy in the classic framework, since the graph has to be partitioned into disjoint sub-graphs forming valid left members for different rules. Matching a sub-graph is itself a difficult operation since it is a graph homomorphism, which is known to be NP complete. The third volume of [14] is entirely dedicated to *parallel GRS*, however this concept is distinct from *distributed GRS* : it considers only how to formally construct a parallel composition of rules which does not directly imply a distributed execution. A distributed execution can be defined using two simple restrictions :

Firstly we consider node-GRS which impose that rules replace only one node at a time. The nodes can themselves be considered as agents doing locally the matching, adding, and gluing. The node's label is called the agent's *state*. The link's label are used by an agent to identify its links. The labels constitute the agent's *context*. The next section introduces a syntax and semantic for *undirected*

node-rewriting rule that tries to capture all what's possible to do locally, with undirected links, and makes the foundation on which to ground SDN.

Secondly, an agent that rewrites must be guaranteed that its neighbors will not, so that they can safely be used as stable anchors for receiving new connections. A GRS which guarantees that two neighbor agents can never be simultaneously ready is called emphneighbor exclusive, where *ready* means that one of the rules is matched (if two rules are matched, one can either define a priority or choose non deterministically).

Definition 1. *A basic Self Developing Network (SDN) is a neighbor-exclusive undirected node-GRS.*

Definition 1 suffers an important drawbacks : it demands that the designer of the rules engineers and proves the neighbor exclusivity. We would like to extend the definition so as to include parallel node-GRS whose formal parallel composition of rule can relax the neighbor exclusive requirement. In order to continue ensuring a distributed execution, we demand that the parallel node-GRS can be simulated by a basic SDN. The definition 6 of simulation enables to see the simulated GRS as a software layer programmed on top of a distributed GRS.

Definition 2. *A higher order SDN is a parallel node-GRS that can be simulated by a basic SDN.*

Execution in a parallel node-GRS. A parallel node-GRS is defined by a parallel composition allowing all ready nodes to update simultaneously. We consider a decentralized update schedule, where each ready node decides whether it rewrites or not, depending on some unknown other factors. A *parallel derivation*, $c_1 \xrightarrow{A} c_2$ from a configuration c_1 to a configuration c_2 simultaneous rewrites a subset A of the ready agents in c_1. This also applies to neighbor exclusive node-GRS since they are a particular kind or parallel node-GRS, where parallel composition is not needed. The configurations of a system are those that can be reached by a derivation starting from the initial configuration. Agents in the initial configuration are called *ancestors*. All the other generated agents are descendants. Some of the ancestors called *hosts* are external agents whose rewriting is conducted externally. Each host is connected to the rest of the configuration through a single connection called *port* which persists during the whole execution. The links labels can be used to store a message for communication, in particular the port label is used for input and output to the hosts. It is important to provide parallel inputs and outputs, so in general, there are several ports.

A parallel node GRS develops a circuit that can be reused and modified indefinitely, depending on the hosts' interaction. In our buffer example, the hosts can push and pop values indefinitely. This generate infinite derivations. Such derivation are implicitly assumed to use a fair update : an agent that is ready cannot be indefinitely idle. If an agent has a choice between some rules infinitely many times, fairness also means that all rules have a non zero probability of being chosen and will be. Alternatively, if the hosts remain idle at some point,

the execution can reach an idle configuration. Such a system can be used to compute a function : the hosts will first input values through the ports, and then output values computed from the developed circuit.

Definition 3. *A parallel node-GRS execution is either an infinite derivation with a fair update or a derivation terminating to an idle configuration*

Cleaned up development. Consider an agent with a loop connection to itself : this agent is its own neighbors, when it rewrites, it can create new agent with connection to itself. Since it is itself deleted, those connections will be left dandling. To avoid this problem, we consider only loop-free execution.

Agents with no connections are useless, because they cannot influence the input/output behavior of the machine. More generally, the network as a whole should remain connected.

Proposition 1. *A dynamic agent network always remain connected, if its initial configuration is connected, and for each rules, the graph formed by neighbors of an updating agent, plus the created agents and connection is connected.*

3 Undirected node rewriting rule.

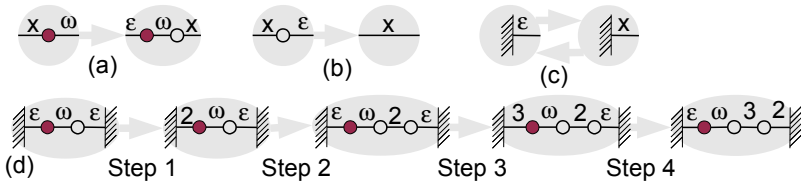

Figure 1. An undirected development rule implementing a buffer ; $x \in \mathbb{N}$, ϵ and ω are special labels. (a) Root agent (disc) (b) data agent (circle) (c) Host read, and write (electrical ground) (d) Example of execution.

We called node-rewriting rule, a rule of node-GRS, that rewrites graph nodes by nodes, independently. In this section we make precise a syntax and a semantic for undirected rules refereed to in definition 1, for network with undirected links.

Firstly, an agent must decide independently whether it matches a given rewriting rule or not. The decision should be *local* in order to be distributed : thus we forbid agents to access the state of its neighbors. The agent's local view includes its own state *qinQ* (node label), and the labels $l \in L$ of its connections. The rule's left member is specified by a multiset of labels $C \in \mathbb{N}^L$. A multiset is a set with repetition, and can also be viewed as a function $\dot{C} : L \mapsto \mathbb{N}$ where $\dot{C}(l)$ is the number of occurrence of l. A rule with left member C is matched, if for each label l, the agent has at least $\dot{C}(l)$ l-neighbors.

The right member includes the number n of agents to be added, their states $q_1, \ldots, q_n$, and new connections specified by triplet $(l_{\text{new}}, v_1, v_2,)$, giving the connection label l_{new} and the two extremities v_1, v_2 (unordered pair) which can address either a newly created agent, or a neighbor. Newly created agents are addressed by an index in $< 1 \ldots n >$, and neighbors by the label of their connection $l \in L$. However there can be several l-neighbor having the same label l on their connections, so the agent must make a prior "*individual bindings*" to distinguish them. It knows that there is at least $\dot{C}(l)$ of them, it chooses randomly $\dot{C}(l)$ connections labeled l and attribute a distinct index $j = 1, \ldots, \dot{C}(l)$ to each of them. The selected neighbors are addressed by indexed labels, the set of thos is noted $C\ \hat{C}$.

There exists a third addressing mode called "*Collective binding*" that binds all the remaining not individually bound l-neighbors, for a given l. It creates multiple connections to each of them, using a single connecting triplet. It can be used only for one extremity in a connecting triplet, and creates multiple connections to each of the collectively bound l neighbors. If it was used for both extremities, the number of created links could be quadratic with respect to the degree of a node. For example an agent with 10 links labeled l_1 and 10 other links labeled l_1 could create 100 connections with a connecting triplet $(l_{\text{new}}, l_1, l_2,)$. Collective binding alone leads to deterministic rewriting. In summary, v_1, v_2 can be either an integer in $< 1 \ldots n >$ that addresses a newly created agent, an indexed label in $\hat{C}$ that addresses an individually bound neighbor, or a label l, that addresses all the not individually bound l-neighbors.

Definition 4. *An Undirected node rewriting rule is* $q_0, C \rightarrow q_1, ..q_n, c_1, ..c_m$ *where* $C \in \mathbb{N}^L, q_i \in Q, c_j \in L \times (< 1 \ldots n > \cup \hat{C} \cup L)^2$

The role of this definition is formal : we will always describe our rules using a more readable graphical notation. Fig. 1 (a)(b) represents an undirected development rule implementing a buffer, it uses only individual binding for the moment being. The left member locates bound neighbors by placing them around a light gray disc showing the label of bound connections, the right member reproduces the same disk, and assumes the bound neighbors conserve the same location on that disc. The buffer agents have two states : root and data. The root-agent updates by inserting a data-agent which stores an integer data item on its input link. Data-agents update by suppressing themselves to make their item available for reading. The root is ready when it has an integer on one edge and the markup ω on the other edge, which distinguishes right from left. A data-agent is ready when it has an integer on one edge, and the empty label ϵ on the other. Fig. 1 (d) shows an execution. The agents are organized in a line starting with the writing host, followed by the root, data-agents, and the reading host. The buffer initial configuration needs to contain one data-agent. The buffer has no capacity limit, since the creation of new data-agents augments the available memory on the fly. Only the two extremities can be simultaneously ready, thus enabling to read and write in parallel, while enforcing neighbor exclusion. This proves that the system is a basic SDN. At step 2 and 4 the number 2 and 3 are stored by data-agent

into to the buffer, the number 2 is retrieved at step 4, and can be read. In our simulation, we will use the following property :

Proposition 2. *Basic SDNs are confluent.*

Because undirected rule do not access the state of neighbors, neighbor exclusivity ensures confluence : the output of a given derivation does not depend on the schedule of rule applications. If the system is mono ancestor, a derivation is summarized by a lineage tree whose root is the ancestor, and branch nodes correspond to all the agents that were generated. Leaves contain either an agent present in the final configuration, or an agent that was deleted. All the agents generated can be uniquely identified by the path leading to them in the lineage tree. If the system is not mono ancestor, we can also obtain a lineage tree by considereing an ad hoc rule that generates the initial configuration from a single ancestor, including the hosts. The branch corresponding to the host is a degenerate tree, it is the sequence of input or output rule issued by the host. Two different derivations are equivalent if their lineage tree is the same. This holds for finite, as well as infinite derivation. In a lineage tree, the number of sub-trees of a given node corresponds to the number of new agents created upon rewriting. For a finite set of rule, it is upper-bounded by a constant since each rules generates a fixed number of agents.

4 Directed node rewriting rule

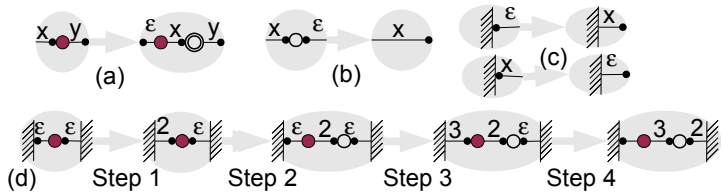

Figure 2. Buffer implemented as a directed system. (a) Root agent's rule(b) Data agent's rule (c) Host's rule (d) Example of execution.

We now consider more generic parallel node-GRS that include a mechanism for parallel composition, so that neighbors can simultaneously rewrite. Such a mechanism can be based on a "*Mutual Connecting Agreement*" , which specifies how to interconnect the agents produced by two neighbors that are simultaneously rewriting. We define such an agreement by orienting the links and deciding that the *owner* of the link is the agent at the source, while the target agent is considered to be *pointed by the link*. An agent canNOT modify incoming links since it does not own them. NOT modify means not create connections to the neighbors, and keep the link as is. It is achieved by preserving the updating

agent instead of deleting it, and let it keep all its incoming links. The updating agent can then be used as a stable gluing point for the neighbors owning a link to it, and updating simultaneously. By convention, links which are owned but are not bound in the left member, will also remain on the persisting agent. This make definition of rules more compact since we are no longer obliged to describe the whole label context.

Definition 5 (Directed node rewriting rule). *Like an undirected rule, except that the extremities (v_1, v_2) in a connecting triplet are ordered, the left member includes a subset of labels $E \subset L$, and a persisting agent is identified.*

The rule application works as follows : Created connection are oriented from v_1 to v_2. Since an agent can modify only the output links, only neighbors connected through output links are considered for binding. However, an agent can test the absence of input links, which is useful for synchronization. The subset of label $E \subset L$ specifies the labels of the input links that if present, will block the updating. If $E = L$ the rule is called *owner-all* and can suppress the agent itself, because no input link needs to be preserved. If all rules are owner-all, the system itself is called *owner-all* and become neighbor exclusive.

The link orientation leads to a more concise higher level representation : In fig. 2 the buffer is programmed as a directed node-GRS, with a single ancestor and no ω markup. Ownership is represented using a tiny black disk at the source, and the persisting agent is identified by an extra circle around it (fig. 2 (a)). Note that in this figure, the link labeled y is only indicated for clarity, it can be omitted since it is an input link and will be kept by the persisting agent, by definition.

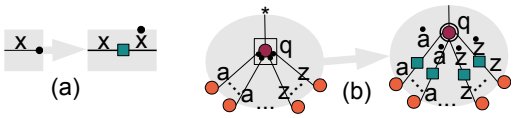

Figure 3. The bipartite transformation (a) Insertion of an edge-agent on each edge. (b) Representation as node rewriting rules.

5 Simulation of Directed node-GRS by basic SDN.

We now want to prove that directed node-GRS are higher order SDN. According to definition 2, we must provide a distributed simulation, we first need to define what is a simulation between parallel node-GRS.

Definition 6. *A simulation of a parallel node-GRS S by another S' if given by a mapping ϕ from the configurations of S to those of S' such that for any parallel derivation $c_1 \to c_2$ in S there exists an image derivation $\phi(c_1) \to *\phi(c_2)$ in $\mathcal{S}'$.*

The mapping ϕ is called a morphism. As usual, $\rightarrow *$ is the transitive closure of $\rightarrow$. If x is a derivation $c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_k$ in S, ϕ maps it to a derivation $\phi(c_1) \rightarrow *\phi(c_2) \ldots \rightarrow *\phi(c_k)$ in S' noted $\phi(x)$, this can be extended to infinite derivation such as executions. Since we consider non deterministic system, we also want to make sure that that for any execution x' in S', there exist an inverse image execution x in S, verifying $\phi(x) = x'$. We call that property *faithfulness*.

We use morphisms called *transformation* that are themselves parallel node-GRS, such as the bipartite transformation shown in fig. 3, which inserts an edge-agent on each oriented connections. The number of rules is infinite though, because each neighbor needs to be individually bound. There is exactly one rule for each possible neighborhood. If t is a transformation and c a configuration, the image $\hat{t}(c)$ is obtained by applying the transformation rule in parallel on every agents of c. If a is a single agent of c, $\hat{t}(a)$ designates the agents and connections generated by rewriting a, and is called the *support* of a. The support of each agent usually contains a distinguished agent called the *master* that sends command to all the other agents of the support, called *slaves*. For example, the slaves are the edge-agents in the bipartite transformation. Only the master's rule depends on the simulated node-GRS, while slaves execute a fixed rule. Master slave transformation can in fact be considered has simply programming : the simulating node-GRS is a software layer (a set of Macros) put on top of the simulated node-GRS.

Theorem 1. *Directed node-GRS are SDNs.*

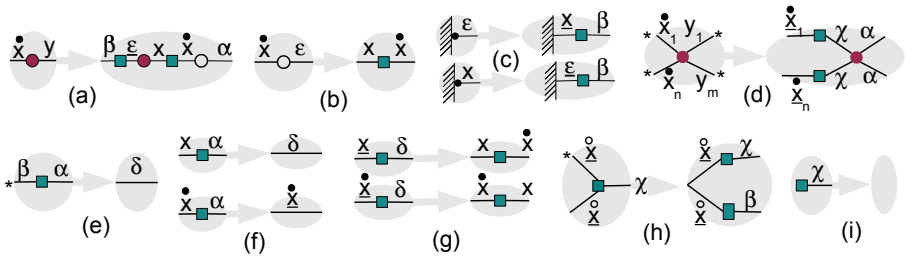

Figure 4. Simulation of the directed buffer. (a,b) recoding of buffer root and data agents (c)recoding of host's rule (d)recoding of NOP rule (e,f,g,h,i) edge-agent's fixed rule.

Proof : Let S be a directed node-GRS, and c_1 a configuration of S. We define a simulating basic SDN S' as follows : For each label l of S, S' needs four labels noted $l, \dot{l}, \underline{l}, \underline{\dot{l}}$, plus four new labels denoted by Greek letters $\alpha, \beta, \chi, \delta$. The morphism $c_1 \mapsto \phi(c_1)$ is the bipartite transformation. An edge agent has two links, one to the owner, and one to the output agent. The original label l is copied on both links, but with a dot above it ($\dot{l}$) on the link towards the owner, in order to encode the orientation. One step of parallel derivation $c_1 \xrightarrow{A} c_2$ in S

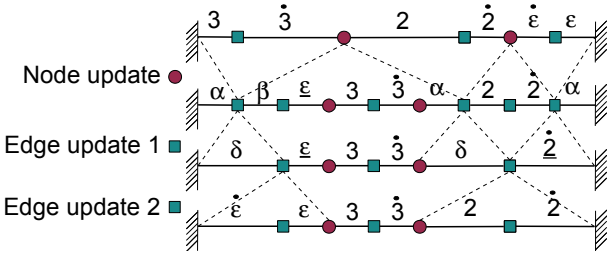

Figure 5. Simulation of the execution step 4 of figure 2 (d)

is simulated in two phases : In phase 1, node-agents execute a recoded version of the original rule using edge-agents as gluing points. fig. 4 (a) (resp. (b),(c)) shows the recoded rule for the buffer's root (resp. data agent, host). For each created connections labeled l, the recoded rule inserts a new edge-agent. If l connects a neighbor to a new agent,(resp. two neighbors or two new agents) the labels of the two links are $(\beta, \underline{l})$ (resp . $(l, \underline{\dot{l}})$). The label of preserved input connections is forgotten and replaced by α. In phase 2, the edge-agents rewrite in two steps. First one edge-agent per connections is restored with rule (e) (resp.(f)) if the connection was created between a neighbors and new agent (resp. between two neighbors). Second the label coding the orientation is restored with rule (g). The role of underlined label $\underline{l}, \underline{\dot{l}}$ is to prevent a node-agent firing before all its edge-agents are done, which is the key that ensures a neighbor exclusive execution.

In phase 1, all the node-agents must update, including those representing idle agents (either because they are not ready, or because they are not in the set A of the considered transition $c_1 \xrightarrow{A} c_2$.) which must execute the recoded version of the NOP rule (d). This rule must be valid for arbitrary many neighbors, it has to use collective binding, which is represented graphically using the '$*$' symbol. The recoding of collective binding on owned link needs a new label χ. The simulation does an iterative processing to insert arbitrary many edge-agents, this is realized by rules (h) and (i) managing χ. The circle above the labels means that it can be either a doted, or not doted label. Rule (i) ends the iteration, and has lower priority than (h).

Last, we must prove faithfulness. Consider an execution x' in the simulating system. It has to be infinite, since agents can always execute the NOP rule. Because of proposition 2 and the fact that the execution is fair, we can reorder the derivation to complete the current simulation step before starting the next. We obtain an equivalent execution x'' so that $x'' = c'_1 \to *c'_2 \ldots \to *c'_k \ldots$ and c'_i is the bipartite configuration obtained after the i^{th} simulation step, and lies in the image of ϕ. Let $c_i = \phi^{-1}(c'_i)$ be the inverse image, $x = c_1 \to *c_2 \ldots \to *c_k \ldots$ is a derivation in S, and by construction, $\phi(x) = x''$. We need to prove that x is an execution. We can have either $c_i \to c_{i+1}$ or repetitions : $c_i = c_{i+1}$ which can be simplified. If the resulting sequence remains infinite, it is an execution (its fair update is implied by the fair update of x''), otherwise their exists a k such that $c_i = c_k$ for $k > i$, which means that node-agents in x" always do the

recoded NOP rule after c'_k. In this case, fairness implies that they could not fire other rules which proves that none of the agent in c_k are ready, and c_k is indeed idle.

6 Examples of Higher Order Self Developing Networks

The definition 2 is generic enough to include other high level formulations. It also allows to classify well known parallel node-GRS, as specific type of SDNs.

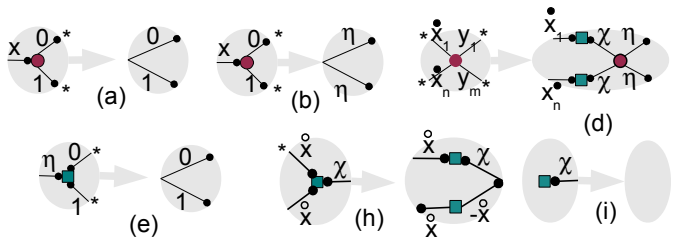

Figure 6. Simulation of transitive directed systems (a) Example of transitive rule (b) its encoding (d) encoding of the NOP rule (e) (h)(i) edge-agent's rule. The notation $-l$ permutes dotted and non dotted label, $-\dot{l} = l, -l = -\dot{l}$

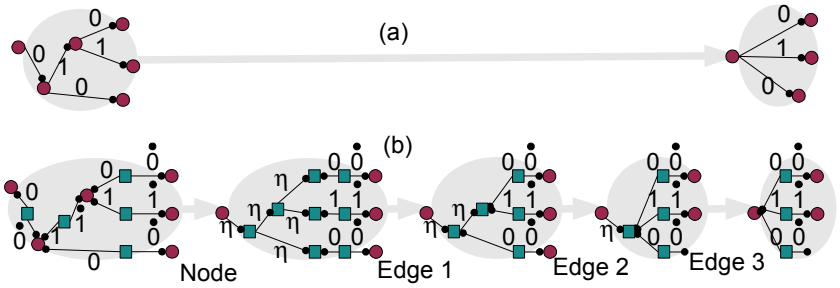

Figure 7. Simulation of one step of execution collapsing a tree of depth 2.

6.1 Other ways of defining SDNs.

Redirecting systems. The input connections can be redirected instead of preserved on a persistent agent, the neighbor at the other extremity will still not perceive any difference.

Definition 7 (Redirecting development rule). *Like a directed rule, except that input neighbors can also be bound but must be used exactly once in a connecting triplet, with preserved label and orientation.*

Redirecting systems contains directed systems. To distinguish input from output neighbors, the context (and binding) is specified using oriented labels $\overrightarrow{l} \in \overrightarrow{L} = L \times \{0,1\}$, which includes a label plus a Boolean coding the orientation of the matched link. The condition on bound input neighbors says that input connections are redirected. Input connections can be redirected through individual or collective binding. Redirection can either be *local* towards a newly created agent, or *transitive* towards a neighbor. Input links cannot be redirected towards another input neighbor, since that would change it to an output neighbor. A cycle in the chains of redirection is forbidden, the redirection should organize in a set of tree, the effect of redirecting is to collapse the tree by gathering all the leaves on the root. An example of transitive redirection is shown fig. 6 (a), and its effect in case of parallel update in fig. 7 (a).

Proposition 3. *Transitive node-GRS are SDNs.*

Proof : We simulate transitive node-GRS using directed node-GRS. A simulation by a basic SDN can then be obtained by composition with a simulation of directed node-GRS by basic SDNs. The transformation is the bipartite transformation, node-agents have ownership of links. The simulation is neighbor exclusive, it uses an owner-all update. One step of parallel execution needs two phases : In phase 1, node-agents execute a recoded version of the original rule using edge-agents as gluing points. Like the recoding used to prove theorem 1, a new edge agent is inserted on each created link, with dotted label to represent direction. The difference is that now, no underlined label are needed to enforce exclusion. The simulating system needs two extra new labels χ and η : As before, χ is used to implement an iterative processing needed for collective binding, while η is used for redirection. In phase 2, edge-agents rewrite using a fixed rule fig. 6(e) which collapses chained transitive redirection in a number of steps equal to the depth of the tree.

Programmed orientation Above the orientation of links used to encode ownership, one sometimes need to encode a *programmed orientation*. Consider for example a chain of agents such as the one used for the buffer. A programmed orientation can be used to distinguish left from right. It is simulated using two "opposite" label (for example L and R for right and left), which are systematically negated when ownership is flipped. When the link is owned, the orientation reads directly, but if the link is not owned, the opposite value must be taken.

Labeling of link extremities Those labels can be read, and modified, only by the agent at the corresponding extremity. They are useful as a local memory per links, and makes it easier for an agent to manage its links without interfering with the neighbors. For example, an extremity flag can be used to distinguish the father from the children in a temporary tree structure, or to locally number the links. If a link is duplicated from one extremity, the extremity labels at the other extremity get duplicated together with the link. Local labels can be incorporated

directly in the simulation of directed systems, they will label only one of the two edge of each edge-agents, the one corresponding to their extremity.

Mixed orientation In mixed graph, links can be either directed, or not. In the same way, in a mixed SDN, ownership can be left undecided. Ownership is no any more represented as an orientation of the link, but using extremity boolean flag which independently labels each of the two extremity of a connection, by 0, or 1. The possible values for the pair of extremity of one link are $(0,1),(1,0),(0,0)$. In the case of $(0,0)$, the link is *not owned* by neither of its two attached agents, which cannot modify the link, except for a very specific modification which consist in acquiring ownership by setting the ownership flag. A link cannot be owned at both extremity, thus $(1,1)$ is forbidden. If by chance ownership is acquired simultaneously at both extremities, a random choice is made. This is a natural way of introducing non determinism in a SDN, useful to break symmetries.

6.2 Classification of existing SDNs.

Ever Growing Network EdNCE graph grammar introduced by Engelfriet and Rozenberg [14] describes a sub class of directed systems using only collective bounding, and forbidding creation of connections between neighbors. Thus if two nodes are not connected, they will never be in the future. When an agent get removed, such as the data agents of the buffer, connecting neighbors together is indispensable to maintain the connectedness. EdNCE grammars develops only ever growing structures. Existing family of graph grammars, such as the algebraic approach or hyper edge replacement grammar [14] allows unconnected nodes to become connected, but they are not designed for distributed execution.

Acyclic network L-systems are grammars introduced by Lindemeyer [10] to model the development of algae and plants. The object being rewritten is a string using brackets representing a compact encoding of a tree. It can be seen as a SDN, where the network is acyclic : in other words it is a tree. In the simplest case of context free L-systems, each agent rewrites independently in parallel. A simulation is done using edge-agents to synchronize the rewriting of all nodes. In the general case, context sensitive L-systems still rewrites agent-wise, but check the state of neighboring agents before applying a rule. For example, this can model a flow of nutrients. The simulation must use an intermediate step of communication so that each agent reconstruct internally the local subgraph composed of one neighbor agent towards the trunk, and several neighbor agents towards the branches. If the context spans more than second order neighborhood, several such steps must be composed.

Network with a fixed number of nodes Self assembly system focus on maintaining a specific subset of link for persistent pairwise communication, or for progressive assembly of a structure, between robots or molecules. As a result, a dynamic network is built and maintained. Klavins [9] use node-GRS to move interacting robots so as to cover a given region. We believe it can be simulated by SDN, if space is accounted for. Rules can create or delete connections (and not agents), and trigger agent movement. This approach has been applied to

interacting robots moving into space, and trying to achieve a particular mission, such as cover a given region while still remaining near each other to be able to communicate using radio signals. Self assembly also models nano-scale mechanism of molecules interacting with each other to create and duplicate macro molecules[13].

Fixed network. If the rules neither creates nor deletes agents or connections, the network is preserved. Such a degenerated self developing system is called "'*static*"' and models a fixed network of finite state automata that can be called *Automata networks*. Automata networks have not been studied very much as a general framework, perhaps due to the difficulty of defining easily a transition for arbitrary neighborhood. Two specific restrictions make it easier to do, and lead to widely studied models : 1-If we restrict the next state transition function to make a commutative, associative reduction of the inputs, and apply a threshold function, this leads to Artificial Neural Networks 2- If we restrict the neighborhood to be the same for every automaton, this leads to Cellular Automata. In both ANNs, and CAs, the connection's label are also fixed, and the agent directly read the neighbor's state. A simulation uses edge agents to copy the agent's state. Static system can hardly be called self developing, since nothing is developed. They model real hardware and allows us to reuse the same machinery of master slave simulation in this context.

7 Deterministic SDN

A GRS is deterministic, if the update of a single agent always gives the same configuration. An obvious source of non determinism, valid for grammars in general, happens when a given context is matched by two rules : the choice between the two has to be random unless a precedence is defined. There is a more subtle source in the case of development. When binding individual neighbors, an agent cannot distinguish between two neighbors, whose connection carry the same label l. Such neighbors are called *local siblings*. The resulting non determinism is avoided only if the network produced by the application of the rules, remain isomorphic when permuting two sibling neighbors. A simple way to obtain this invariance is to use only collective bounding.

When considering the updating of many agents instead of a single one, a last source of non determinism is due to decentralized execution : only a randomly selected subset of all the ready nodes are rewritten at each step. In some cases, we would like to obtain a *confluence* property to ensure that the system consistently converges to the same configuration despite this randomness. Only confluent system have an output that is defined independently from a schedule of update. Since an SDN is a rewriting system, we can apply the general definition of confluence. Because we have agents, the simpler concept of commutativity is easier to deal with.

Definition 8. *A development system is commutative if it is deterministic, and for any two agents a_1, a_2, updating a_1 before, after or simultaneously with a_2 gives the same network.*

Commutative rewriting systems are confluent, so commutativity is stronger than confluence. Basic SDNs are commutative (proposition 2.

Deterministic redirecting systems can be made commutative by adding appropriate delays. Consider an agent a_1 owning a connection c_1 to a_2. Let an update of a_1 creates a connection c_2 to a_2. From a_2's point of view, the label l of c_2 has appeared in its context. We say that c_1 *"produces"* l. If c_2 is not owned by a_2, it can itself produce another label l'. By transitive closure l' is also produced by c_1.

Proposition 4 (Commutative closure). *A transitive node-GRS becomes commutative if agents wait for removal of input links that produce labels bound by its matching rules.*

Proof : Consider again an agent a_1 owning a connection c to a_2, and both a_1 and a_2 are ready. The schedule of a_1 and a_2 's rewriting does not matter, because in all cases, once a_1 and a_2 have finished rewriting, the links created by a_1 through c will all end up on the persisting copy of a_2.

Corollary 1. *The directed buffer is commutative*

Proof : The commutative closure leaves the system unchanged : agents do not need to wait since no link produce any labels.

8 Conclusion

This work first presents a minimal framework in which a distributed graph rewriting can be defined : namely an undirected network, and a neighbor exclusive node rewriting rule. The basic framework is extended by way of simulation, in order to define a formal generic model called Self Developing Network (SDN). We present a particular extension using directed link, that relax the neighbor exclusion requirement The second half of this work [5] further enrich the model using finite state automata, and proves an "efficient" intrinsic universality result in linear time, while a simulation by a Turing Machine needs exponential time. This formalizes the specific power of SDN, and justify considering an unbounded source of processing elements as a worthwhile line of research.

We acknowledge fruitful comments made by L. Maignan, and C. Eisenbeiss.

Références

1. G. Agha. *Actors : a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.
2. A. M. Ben-Amram. Pointer machines and pointer algorithms. Technical Report 95/21, Kobenhavns universitet, Univ. of Copenhagen, 1995.
3. L. Cardelli. Brane calculi. interactions of biological membranes. In *Computational Methods in Systems Biology*, pages 605–614. Springer, 2004.

4. S. A. Cook and P. W. Dymond. Parallel pointer machines. *computational complexity*, 3 :354–375, January 1993.
5. Frederic Gruau. Self developing networks, part 2 : Universal machines. Technical report, INRIA, 2011. submited to the Turing Centenary Conference, 2012.
6. Yuri Gurevich. Kolmogorov machines and related issues. *Bulletin of the European Association for Theoretical Computer Science*, 35 :71–82, 1988.
7. C. Hewitt, P.Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
8. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.
9. E. Klavins. Programmable self-assembly. *Control Systems Magazine*, 24(4) :43–56, August 2007. See the COVER!
10. A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *jtb*, 18 :280–299, 1968.
11. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-verlag, 1980.
12. R. Milner. The polyadic -calculus : a tutorial. In W. Brauer, F.L. Bauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Lecture Notes in Computer Science, pages 203–246. Springer-verlag, 1993.
13. J-P. Patwardhan, C. Dwyer, A. R. Lebeck, and D. J. Sorin. Circuit and system architecture for DNA-guided self-assembly of nanoelectronics. In *FNANO*, pages 344–358, 2004.
14. Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1,2,3. WSP, 1997.