RAPPORT DE RECHERCHE

L R I

SELF-DEVELOPING NETWORK : A SIMPLE AND GENERIC MODEL FOR DISTRIBUTED GRAPH GRAMMARS

GRUAU F

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

01/2012
(revised version 03/2014)

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 : A simple and generic model for distributed graph grammars.

Frédéric Gruau

Laboratoire de Recherche en Informatique, Université Paris Sud

Résumé This work investigates what is the simplest of rewriting graph in a distributed way. Nodes of the graph are identified as independent agents doing themselves the rewriting, hence the appellation "Self-Developing Network" (SDN).

We first study the most general way of rewriting a single node. We define two complementary ways of selecting neighbors based on link labels : an individual random selection allowing to process neighbors one by one, and a collective selection allowing the creation of connections towards arbitrary many neighbors. The resulting node-rewiting rules can be applied in a distributed way, provided two neighbors are never simultaneously ready. This constitutes our first working definition of elementary SDNs.

This neighbor-exclusive requirement can be relaxed using link orientation and adapt the semantic of rule application. The resulting enhanced model can be programmed as a layer on top of elementary SDNs. We finally obtain a definition of SDN that is : 1- simple enough to be considered as canonical 2- not dependent on the particular scheme used to achieve distributed execution 3- generic enough to allow many other expressive notations, and a classification of existing well-known models as specific SDNs.

Les réseaux auto-développants ; un modèle simple et générique pour des grammaires de graphe distribuées.

Ce travail étudie la façon la plus simple de réécrire des graphes de façon distribuée . Les Nœuds du graphe sont identifiés comme des agents indépendants faisant la réécriture eux-mêmes, d'où l' appellation "réseau auto-développant " (SDN) .

Nous étudions d'abord la façon la plus générale de réécrire un seul nœud. Nous définissons deux façons complémentaire de sélectionner les voisin, a partir de l' étiquettes de la connexion qui les relie : une sélection aléatoire individuelle, qui permet de traiter les voisins un par un, et une sélection collective permettant la création de connexions vers des voisins arbitrairement nombreux. Les règles de réécriture de nœuds résultantes, peuvent être appliquées de manière distribuée , à condition deux voisins ne soient jamais simultanément prêt à être réécrit. Cela constitue notre première définition de travail de SDN élémentaires .

La contrainte d'exclusion mutuelle entre voisins peut être relaxée en orientant les connections, et en adaptant la sémantique de l'application des

règle. Le modèle ainsi amélioré peut être programmé comme une couche au-dessus des SDN élémentaires. Nous obtenons finalement une définition des SDN qui est : 1 - assez simple pour être considérée comme canonique 2 - ne dépend pas du modèle utilisé pour réaliser l'exécution distribué 3 - suffisant pour autoriser de nombreuses autre notations encore plus expressives, et une classification des nombreux modèles bien connus comme des cas spécifiques de SDN.

Keywords: Self developing network, massive parallelism, simulation, formal system, graph grammar

1 Introduction : motivation for a new model.

This work takes its inspiration from the amazing parallelism present in biological system made of many cells which exist in space and time, and can update in parallel. The parallelism is available *everywhere in space, constantly through time*. Moreover, cells can divide and create other cells thus increasing the available parallelism through time : 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 iteratively create other cells and exchange messages, until a whole organism is unfolded : in short, the system *self develops*.

We believe there is a gap to be filled in the catalog of formal models describing parallelism : namely, systems where computing is *synonymous of creating exploitable parallelism*. Informally, a Self Developing Network (SDN) consists of a network of decentralized finite-state agents that update independently in parallel. They can modify their state, but also produce other agents and connections thereby "self-developing" the network.

Computing with SDN is *synonymous of creating parallelism*. An SDN not only specifies a network of agents operating in parallel, it also encodes a parallel development of that network which can grow arbitrary large out of a single ancestor agent, and can also shrink back. An agent is very fine grain : it holds a single scalar data, so it cannot do anything meaningful by itself [1], therefore, doing computation is *synonymous* to developing a network. An arbitrary large set of n scalars can be processed, by developing a network of n agents. We have been studying sorting and matrix multiplication in [6]. Each agent also holds rules that will generate appropriate connections between agents, let them communicate their scalar, and do the necessary computation in order to solve the problem at hand.

Computing by developing network is not so difficult. 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 [2] memory is very convenient : data is stuffed

1. When an implementation is considered [6], agents will keep moving through hardware in order to accommodate development, therefore agents need to be light weight.
2. Brave Passive Big Global

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.

Computing with SDN is synonymous of creating *exploitable* parallelism. There exists already many formal parallel models : process algebra, pi-calculus, population protocols...To our knowledge, they all use a *global name space* : a process can communicate with any other, using its name or id. The use of global identifier implicitly requires a shared memory for communication, which is not conducive to scalable parallel performance. In SDNs, communication is local : two agents can communicate only if there exist a connection between them. When generating new agent, connection inheritance is specified. In other words, graph rewriting is done. Since the communication pathways are known at every instant during execution, and are updated in a "continuous way", the whole network of agents can be dynamically mapped on hardware, or more precisely, continuously re-mapped during development. The parallelism will be exploitable, if the developed network match the processor network of the target parallel hardware. For example, it won't be efficient to develop a 3D grid on a 2D mesh. The problem of an efficient parallel implementation of self-development is our long-term project. In [6], we discuss the theoretical foundation, and in particular, how to dynamically map the developed network on hardware.

We have been using SDNs for quite a long time to program real tasks, and have become convinced of their generality. Several existing formalism such as EDNCE graph grammars, L-system or self-assembly system can be considered as restrictions of SDNs. Special-purpose examples of SDNs exists in the literature, a nice example [14] uses SDN to simulate the Von-Neuman self-reproduction. The rules are devised so that while developing, each created agent always has exactly three neighbors, which in turn, allows to bound the possible rule application contexts. The goal of this paper is to give a formal definition of SDNs on which the community can agree. It should be simple and generic. We propose 3 steps :

1. Build Elementary SDNs which are simple to define but requires mutual neighbor exclusion.

2. Use directed link in order to relax the requirements of mutual exclusion. The resulting model can be programmed on top of elementary SDNs. It can therefore be considered as a higher-level SDNs.
3. Add more programming layers, to either improve the expressiveness, or classify existing approaches has specific instances of SDNs.

1.1 Review of similar work

The best way to define more precisely the specificity of SDNs is probably to contrast its differences with existing models. Let us now review well-known models with dynamic creation of agents.

Term rewriting Systems have been used to model and reason about growing distributed systems, however the structure encoded are trees, while SDNs use graph, which leads to a much more expressive power. An SDN is like a circuit where gates have the capability of dynamically adding new gates. The specific shape of the circuit is crucial to organize the computation according to an objective. Another example of distributed formalism : *Population protocols* does not even develop a communication structure between agents, which, we believe, reduces the spectrum of possible computation to only probabilistic tasks.

In *Process algebra*, dynamic creation of agents is a primitive operator just like in SDNs. These family include many models such as the Actor model [7] and [2] and the PI calculus [11], which are useful to prove theorem about distributed algorithm. Brane calculi [4] is used for simulating a single biological cell, where structure is light : within a cell sub compartment, any molecules can interact with any other... All of these models use a *global name space* whereas in SDNs the network used for messages communication is explicitly instantiated at run time.

Structurally Dynamic Cellular Automata (CA) [1,10] allow to modify the connections between the CA cells, according to a rule similar to the state transition rule associated with the conventional CA. To create or delete a connection, the rule use the preexisting, initial lattice connectivity of CAs. Unlike SDNs, agents are not added dynamically. An infinite number of agents is present in the initial configuration.

In chemical inspired systems, agents are molecules that interact pairwise with each other and generate new molecules. Such model have been used in [3] to maximize the parallelism in the description. Their real distributed execution is not scalable though, since it needs an implicit heavy centralized process that can detect the end of a reaction by checking that no more interactions are possible for every pair of molecules.

2 Elementary Self Developing Networks

Self Developing Networks (SDNs) consider networks of agents which can communicate between local neighbors, and create new agents. Upon creation, connection are also inherited locally : a created agent is connected to neighbors of

its parent agent. The formal tool to modify a graph in such a local way, is known and studied as Graph Rewriting rule Systems [13] (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
2. Removing this sub-graph, adding the right member, and gluing it to the rest of the graph.

The third volume of [13] is entirely dedicated to *parallel-GRS*, however it considers only how to formally construct a parallel composition of rewriting rules. By contrast, SDNs can be informally defined as *distributed GRS* which consider a distribution of the network onto several processing elements and would like to execute concurrently several rewrites concerning different parts of the network. In general, distributed rule application involves a difficult *partition problem* : the graph has to be partitioned into disjoint sub-graphs forming valid left members for different rules. First, matching a sub-graph is itself a difficult operation since it is a graph homomorphism, which is known to be NP complete. Second, there may be many possible way of partitioning.

Section Outline We will defined SDNs rewriting rules as the minimum number of restrictions to add to generic graph rewriting rules in order to obtain a distributed-GRS. We are interested by the computation that can be done by the developing network, we therefore have to define the inputs and outputs. Finally we introduce elementary SDNs wich just need an additional constraint of mutual exclusion ensuring a decentralized execution.

2.1 Node rewriting Rule

As a first requirement, we impose that the replaced subgraph is reduced to a single node, such rules are usually called *node-rewriting rule*. Node rewriting greatly simplifies the partition problem, we know that each partition contains exactly one node, what is left to do is partitioning the edges. We will present two equivalent method for that purpose : by mutual exclusion, and by link orientation. More importantly, with node rewriting, the network being rewritten becomes like a "growable parallel hardware". We can imagine that the nodes are independent agents doing locally the matching, adding, and gluing in a distributed way ; hence the appellation "self developing network" [3].

Some examples of Node-rewriting rule such as EdNCE graph grammar [13] have been proposed in the literature of graph grammar. In this paper, by node-rewriting we mean the most generic rewriting that can be done by a node, while still allowing for a distributed execution between different nodes. Consider an agent a in the process of creating n new agents indexed by $i = 1 \dots n$. We now need to answer the following question : how can a specify new connections

3. Of course, such a growable hardware does not exist, except for biological systems. We motivate SDN as a new programming model that delivers exploitable parallelism.

between those n new agents, and also between the former neighbors of a, in the most generic way ? Let $q \in Q$ be the node's label also called the agent *state*, and let $l \in L$ be the connection's labels. Because we want agents to be independent, our agent a cannot access the state of its neighbors. It's local view, is therefore the set $C(a)$ of its connections labels. Since a given label can appear several times for distinct connections, $C(a)$ is in fact a multi-set of labels. An agent cannot distinguish between two neighbors connected via links carrying the same label l, which we will call l-neighbors. Let $|a|_l$ be the number of l neighbor for a given label l. In order to be able to bind neighbors individually, whenever $|a|_l > 1$, a needs to do a prior non-deterministic indexing of labels $l_i, i = 1 \dots |a|_l$ of its l-neighbor.

New connections are specified using a *connecting triplet* $(l_{\text{new}}, v_1, v_2,)$, with the new connection label l_{new} and the extremities v_1, v_2 which can be either of :

1. A newly created agents, specified by its index in $\{1 \dots n\}$
2. An *individually bound* neighbor specified by an indexed label $l_i, i \in 1..|a|_l$.
3. All neighbors having a given label $l \in L$, and not individually bound, in which case, up to $|a|_l$ connections can be created simultaneously.

The third connecting mode is called *collective binding*, and allows to handle arbitrary large context, with a finite list of connecting triplets. Such a binding is necessary, because the number of neighbors for a given agent can grow arbitrary large through development. Collective binding can be used only for one extremity v_1 exclusive v_2 in a connecting triplet $(l_{\text{new}}, v_1, v_2,)$. If it was used for both v_1 and v_2, the number of created connections would be quadratic with respect to the degree of a node. For example an agent with 10 connections labeled l_1 and 10 other connections labeled l_2 would create 100 connections labeled l_{new} with a connecting triplet $(l_{\text{new}}, l_1, l_2,)$. In the following definition, $\widehat{C}$ is the set obtained from C by attributing a unique index to the different occurrences of a given label. For example $\widehat{\{x, x, y\}} = \{x_1, x_2, y_1\}$.

Definition 1. *A node rewriting rule is given by* $(q_0, C) \to (q_1, \dots q_n, c_1, \dots c_m)$ *where* $C \in \mathbb{N}^L, q_i \in Q, c_j \in L \times (\{1 \dots n\} \cup \widehat{C} \cup L)^2$

An agent can fire this rule, if its state is q_0 and its context contains C. It apply the rule by indexing its neighbors, deleting its connections, creating n new agents with state $q_1, \dots q_n$ and creating connections according to each connecting triplet $c_1, \dots c_m$. The goal of this definition is just to give a precise formalization, at one point in this paper. This concept of node-rewriting rule can be applied to different network structure. In this paper we consider directed network and then undirected networks. For designing examples of rules, we will always use an intuitive graphical convention, that avoid the use of indexes. We will consider two kinds of collective binding : one in which at least one neighbor must be present, and the other one where there can be zero neighbors, represented using respectively the symbol '+' and '*', a common notation of formal language theory. The '+' form is a syntactic sugar, easily encoded using the '*' and an individual binding.

How do we interconnect the agents produced by two neighbors which are simultaneously rewriting ? We will see different versions of node-rewriting rules, depending on how we solve this problem. The most simple way is just to make the hypothesis that it never happens, because the programmer carefully designed the rule with that purpose in mind, and can actually prove that whatever scheduling takes places, two neighbors are never simultaneously ready to rewrite.

2.2 Undirected Node rewriting Rule

As we search for the simplest model of self development, it makes sense to consider the simplest network structure : undirected labeled graph.

Definition 2. *An undirected node rewriting rule is a node rewriting rule acting on undirected networks*

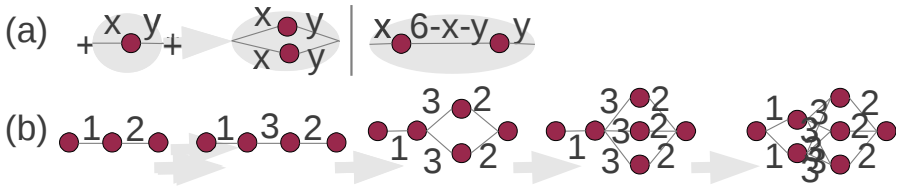

Figure 1. node rewriting rules Developing all serie-parallel graphs : (a) Rules for parallel and serial creation. The symbol '+' codes for collective binding, with at least one link present. (b) Example of execution.

For such rules, in a connecting triplet $(l_{\text{new}}, v_1, v_2,)$, the order between v_1 and v_2 does not matter. Figure 1 describes an undirected rule which develops any so-called *s*erie-parallel graph. The rules are described with the following graphical convention : 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 rule in fig. 1 (a) uses three integer labels $x, y \in \{1, 2, 3\}$. In the left member, x and y can be any of these three numbers, with $x \neq y$. An agent rewrites into two agents, in two possible ways :

1. In the "parallel" rewriting, all the links are duplicated, one copy for each new agent.
2. In the "serial" rewriting, one agent get x-links, and the other y-links. A link with label $6 - x - y$ is added to connect them.

The label $6-x-y$ is the third possible label $z \in \{1, 2, 3\} \setminus \{x, y\}$. It ensures that any generated nodes will always have exactly two of the three possible labels within their neighbors, exept the extremity nodes which have only one, and cannot rewrite. All the agents have the same state, which can be ignored. The SDN

is *ever-growing* : development never stops, network size always augment. The figure shows a development starting from an initial network of three nodes. The central node does one serial division , one offspring does two parallel divisions, and then the other does one last parallel division during which the three links labeled 3 are collectively bound and get duplicated.

2.3 Providing input-output with grounded node-GRS.

A *node-GRS* designates a rewriting system including a set of node-rewriting rules plus an initial network whose agents are called the "*ancestors*", all the other generated agents being the *descendants*. In order to define a computation, we need to "ground" the node-GRS with ports carrying the inputs and the outputs. We define *hosts* as designated ancestors which remain present during the whole execution. Connections to the hosts are called *port* and also persist, thus the number of hosts and ports is a constant of a given node-GRS. The port's labels is used as a memory shared by the host, and the developing network. Reading and writing this label amounts to input and output values as shown in fig. 2 (c_1,c_2).

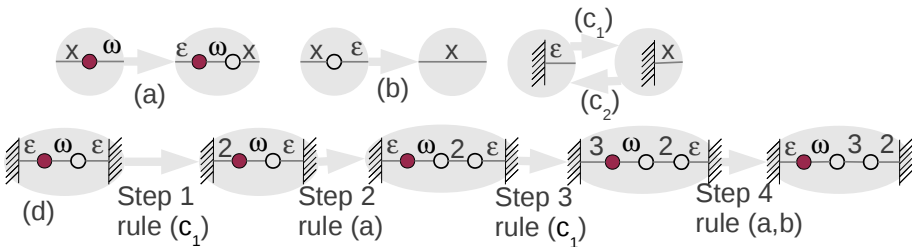

Figure 2. A grounded node-GRS implementing a buffer ; Hosts are drawn as an *electrical ground*. $x \in \mathbb{N}$, ϵ and ω are special labels. (a) Root agent (disc) (b) data agent (circle) (c) Host read, and write (d) Example of execution.

Fig. 2 (a)(b) represents a grounded node-GRS implementing a buffer. The buffer uses only individual binding. Buffer agents have two states : root and data. Fig. 2 (d) shows an execution. The agents are organized in a line starting with a writing host, followed by one root, several data-agents, and a reading host. The buffer initial configuration needs to contain one data-agent.

An agent is called *ready* if at least one of the rules is matched ; [4] 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. The root-agent updates by inserting a data-agent which stores an integer data item on its left connection. Data-agents update by suppressing themselves and make the next data item

4. If an agent match two rules or more, one can either define a priority on rules or choose non deterministically.

available for reading. The buffer has no capacity limit, since the creation of new data-agents augments the available memory on the fly. 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.

The buffer example illustrates that a developed network has an inner state, and can be reused and modified indefinitely, depending on the hosts' interaction : The hosts can push and pop values indefinitely and generate infinite derivations. Alternatively, if all the hosts remain idle at some point, the execution may reach an idle configuration, where no further rewriting is possible. Such a system can be used to compute a function : the hosts will first input values through the ports, and then retrieve values computed from the developed circuit. The buffer uses a single input and output port. In general, one should provide many ports for parallel inputs and outputs, otherwise the parallelism inherent in the developed network cannot be exploited.

2.4 Simple Definition of Elementary SDNs

In a decentralized distributed execution, at a given time t, any agent which is ready (i.e. which context matches a given rule's left member) may decide to rewrite. Thus, a network configuration c_1 develops into c_2, by rewriting a subset A of the ready agents. This parallel rewriting step is noted $c_1 \xrightarrow{A} c_2$.

A node-GRS for which two neighbor agents can never be simultaneously ready is called *neighbor-exclusive*. Such system can be executed in a decentralized distributed way, because any agent that rewrites is guaranteed that its neighbors will not, and can safely be used as stable anchors for receiving new connections.

Definition 3. *An elementary Self Developing Network (SDN) is a neighbor exclusive undirected node-GRS.*

The serie-parallel GRS is not neighbor exclusive, the two agents doing the parallel division are adjacent, and simultaneously ready. Should the two agents decide to divide simultaneously, we would not know what to do. so decentralized execution is not defined.

The buffer GRS is neighbor exclusive : only the input host and either the output host or an already read data agent can be simultaneously ready, in which case they are separated by the data agent havingThe rule for p seem to bind a not-owned link l, which we said was forbiden. This is a trick used to indicate which is the persisting agent in the right member : the one who get l, which here is the XOR. This notation is coherent, since the persisting agent does inherit not owned links, if the neighbor does not modify them. a label ω on the left. In other words, reading and a writing the buffer can be done simultaneously, while still enforcing neighbor-exclusion. This proves that the buffer is an elementary SDN. Note that a node-GRS can always be made neighbor exclusive using randomness, by adding a rule implementing a random local tournament between simultaneously ready neighbors, and blocking one of the two (or both in case of equality).

Proposition 1. *Deterministic elementary SDNs are confluent.*

Proof : Consider two distinct parallel rewriting setp, and let E (resp. F) be the set of agents involved. Because of neighbor-exclusion, two agents which update simultaneously, are not neighbor, and do not influence each other. Rewriting agents in $F \setminus E$ (resp. $E \setminus F$) will therefore lead to the same configuration, The rule for p seem to bind a not-owned link l, which we said was forbiden. This is a trick used to indicate which is the persisting agent in the right member : the one who get l, which here is the XOR. This notation is coherent, since the persisting agent does inherit not owned links, if the neighbor does not modify them. obtained by rewriting agent in $E \cup F$.

If the elementary SDN 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 considering 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.

3 Higher level Self Developing Network.

Section Outline In our quest for reaching the simplest possible definition of SDNs, we gave the requirement of mutual exclusion. While this is probably a fundamental key for the simplest definition, it is also a bit awkward, since it is the task of the person who design the GRS to ensure mutual exclusion. There exists other ways of specifying development, that does allow simultaneous rewriting of adjacent nodes. In particular, there is a rather natural way to do it using directed links. We will show that it is possible to execute such "directed SDN" using our first simple model, One agent is encoded using several elementary SDN agents, and one parallel rewriting step is decomposed into several mutual exclusive elementary steps. Therefore, directed SDNs can also be considered as occurrences of SDNs. We call them "*Higher level SDNs*". The encoding inserts an edge agent on each edges. The resulting graph is bipartite, edge agents separate node agents, and vice versa. A mutually exclusive execution is obtained by alternating between edge agents and node agents.

3.1 Directed node-rewriting rules

Using oriented connections allows to partition the network naturally : one simply decides that each connection belongs to the node which is at the *source*,

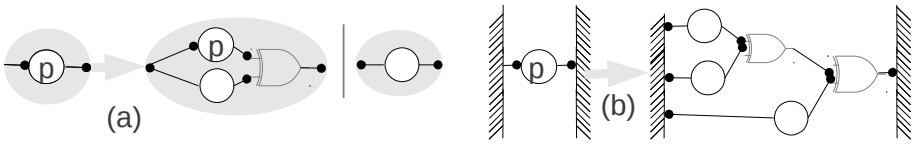

Figure 3. A directed rule developing a parity network. (a) Recursive rewriting of p. empty disc represent idle nodes. (b) Example of execution, with two recursive rewrites.

while the target agent is considered to be *pointed by the connection*. Having partition the network into disjoint left members, one can now perform simultaneous rewriting of adjacent nodes.

In general, simultaneous rewriting of neighbors is possible given a "*Mutual Connecting Agreement*" which describes how to interconnect the agents produced by two neighbors which are simultaneously rewriting. must provide a "*Mutual Connecting Agreement*" which specifies how to interconnect the agents produced by two neighbors which are simultaneously rewriting. Such an agreement, can be defined using directed connections : completed by a specific Mutual Connecting Agreement using the link orientation. The simplest agreement is goes as follows : an agent is not deleted upon rewriting. We must specify connections for it, using connecting triplets, just like other created nodes. The preserved agent a can then be used as a stable gluing point for a given input neighbor, updating simultaneously. Bounded connection are deleted, and new connections specified by the right member of the rule are drawn. Output connection that are not bound implicitly remain on the preserved agent.

Definition 4. *A directed node-rewriting rule is a node-rewriting rule acting on directed networks. Link orientation defines ownership, agents are preserved, and serve as gluing points.*

In comparison with definition 2, the link's label need to be coupled with a boolean encoding wether it is an input or an output link. Moreover, in a connecting triplet $(l_{\text{new}}, v_1, v_2)$, the order between v_1 and v_2 now matters. It indicates the orientation of the created link. Furthermore, the semantic of ownership imposes some constraints on the rule : Since an agent can modify only the owned output connections, only those can be bound in the left member. An agent canNOT modify incoming connections, where NOT modify means keep the connection as is [5], but also NOT create connections to the neighbor. The input link remain on the preserved agent. This is used in our graphical notation, to identify the preserved agent in the right member without having to introduce another markup : the preserved agent is the one who gets all the input links, for example in fig. 3 it is the XOR.

If an agent owns all its connections, it does not need to be preserved and can be deleted. Such a rule is called *owner-all*. If all rules are owner-all, the system itself is called *owner-all* and is neighbor-exclusive. A rule which deletes its active

5. maintain the label and orientation

agent is implicitly owner-all. A rule can also be owner-all for the purpose of synchronization.

Fig. 3 (a) represents a directed rule acting on a node labeled p. It develops a network computing the parity function, i.e. it inputs booleans, and return true if the number of true input is even. This development uses only individual binding. Ownership is represented using a tiny black disk at the owner extremity. The owned links correspond to the inputs of the parity function. Rewriting p is either recursive, or gives an idle node, which is used just for duplicating signals and can be removed in a second step.

Since p has two possible rewritings, the parity rule is not deterministic, it can generate an infinite family of parity networks. Fig. 3 (b) shows an execution with two recursive steps, starting from an initial configuration with one input, and one output host represented as electrical ground. The execution generates a network which computes the parity of three inputs using two chained XORs. If we would want to generate a network for exactly n inputs, we would need to include a loop counter in the agent p, initialize this counter to n, decrement it at each recursion and do the non-recursive rewrite when it reaches 0.

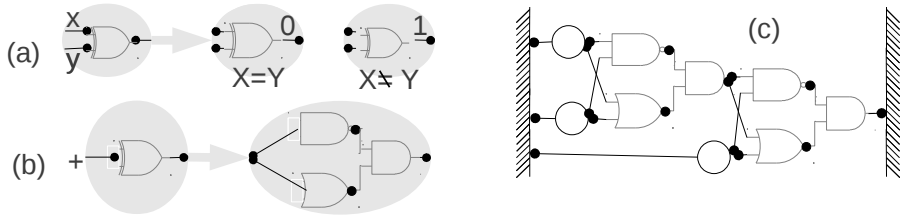

Figure 4. (a) Rule for computing a XOR (b) Rule for developing a XOR (c) Simultaneous development of the two adjacent XORs of fig. 3 (b).

Consider now the generic problem of simulating boolean circuits. Fig. 4 (a) shows how a XOR gate rewrites like a synchronous data-flow gate : It consumes two tokens on its input links, and generates one token on the output link. The link labels encodes a single token encoded by a label $0, 1$ or empty. The link orientation encodes the synchronization : A gate needs to own all its links, before it can fire. It then gives back ownership to the neighbors. Fig. 4 (b), shows how a XOR gate can be rewritten using only OR, AND, and NAND gates. Here collective binding is used : the unique input links of the rule will bind two connections. This reflects the symmetry between the two inputs which are both sent to the NAND gate and to the OR gate.

The parity-GRS shows how a data-flow graph can be developed, using rewriting rules which either create new nodes for development or modify labels and orientation for communication and computation. This “dynamic data-flow graph” still has two problems :

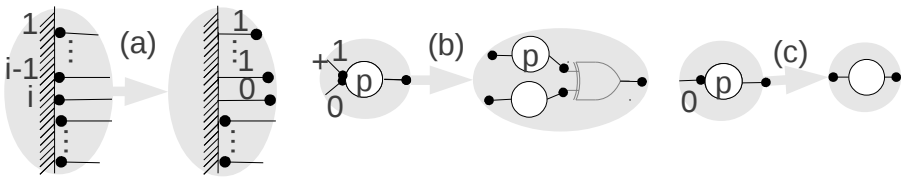

Figure 5. Grounding the parity self-develoment : (a) Bits communicated by hosts 1 $\ldots i$, at step i. (b) Parity Rule preserving ports (c) non recursive case.

1. An agent cannot statically distinguish between boolean inputs and outputs [6]. We do have a link orientation, but it is used for synchronization, and is constantly flipping back and forth. Thus it cannot serve a second purpose.
2. The parity-GRS is not grounded, since the number of input ports increments by one after each recursive rewriting.

The first problem is solved by representing a *p*rogrammed orientation ; using two "opposite" label : for example L and R for right and left. The label is systematically negated when ownership is flipped ; The second problem is solved by letting the input host do extra coordinated communications as shown in fig. 5 (a). At step i, it send 0 on the i^{th} port, and 1 on the j^{th} port, $j < i$. This bit allows the SDN to distinguish the i^{th} port, which can then be moved by the modified parity rule of fig. 5 (b,c). In subsection 4.2 we solve this port problem more elegantly, without having to implement supplementary host communication. The links are arranged in a list naturally, without having to process link indexes and two such list of links can then be put in correspondence.

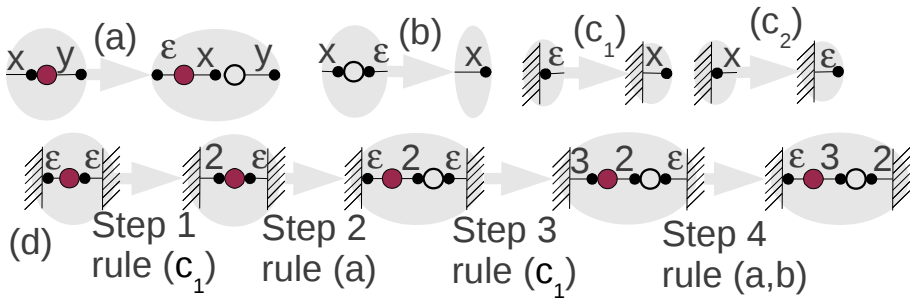

Figure 6. Buffer implemented as a directed node-GRS. (a) Root agent's rule(b) Data agent's rule (c) Host's rule (d) Execution.

Directedness leads to a more concise higher level representation : In fig. 6 the buffer now needs a single ancestor and no ω markup.

6. The figure makes falsely believe that a gate can do that, because of the gate pictorial representation which is not a symmetric circle.

3.2 Simulation of directed rules by undirected rules

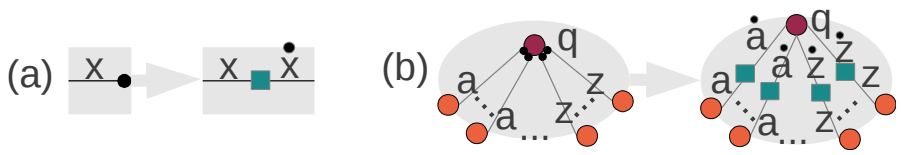

Figure 7. The bipartite transformation (a) Insertion of an edge-agent on each edge. (b) Representation as a directed node-GRS.

The main result of this paper states that directed node-GRS can be considered as higher-level SDNs : they can be programmed on top of undirected node-GRS. The converse is shown in [5], therefore the two formalisms are really equivalent, and the definition of self-development does not depend on the particular choice we make, to enable distributed rewriting.

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

Proof : Let S be a directed node-GRS. We program it as an elementary SDN $\phi(S)$ as follows : For each label l of S, $\phi(S)$ needs four labels noted $l, \dot{l}, \underline{l}, \underline{\dot{l}}$, plus four new labels denoted by Greek letters $\alpha, \beta, \chi, \delta$. We encode a directed network as an undirected one, by inserting an agent called *edge-agent* on each directed link labeled l, as shown in figure 7 (a) An edge-agent has two connections : one to the owner, and one to the output agent. The original label l is copied on both connections, but with a dot above it ($\dot{l}$) on the connection towards the owner, in order to encode the orientation. The original agent itself remains untouched and is called *node-agent*. Note that this transformation amounts to do a single rewriting step of a directed node-GRS, as shown in fig. 7 (b), so self development can be used also for encoding.

One step of parallel rewriting in S $c_1 \xrightarrow{A} c_2$ is simulated in three steps of paralle rewriting in $\phi(S)$, illustrated in fig. 8.

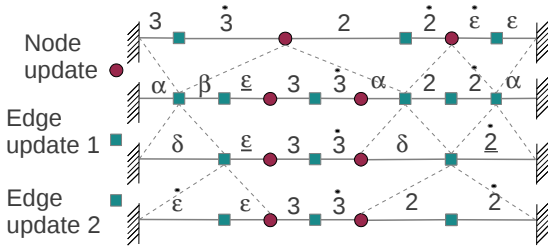

Figure 8. Execution of the buffer simulation for step 4 of figure 6 (d)

1. Node-agents add one edge agent on each created connections.
2. Edge-agents simplify and restore one edge-agent per connections
3. Edge-agents restore the labeling corresponding to the bipartite encoding.

Step 1 in $\phi(S)$: A node-agent is called a *master*, because its rules are compiled from the simulated system S, as an example, fig. 9 (a) (resp. (b, c_1 and c_2)) shows the compilation of the buffer's root (resp. data agent, host read and write). The links to edge-agents corresponding to not owned connection, are preserved and labeled by α. For each created connections labeled l, a new edge-agent is inserted with labels $(\beta, \underline{l})$ or $(\beta, \underline{\dot{l}})$ [7] (resp . $(l, \dot{l})$) if l connects a neighbor to a new agent, (resp. two neighbors or two new agents). The label β is inserted towards the neighbors, so that it will always pair with an α link.

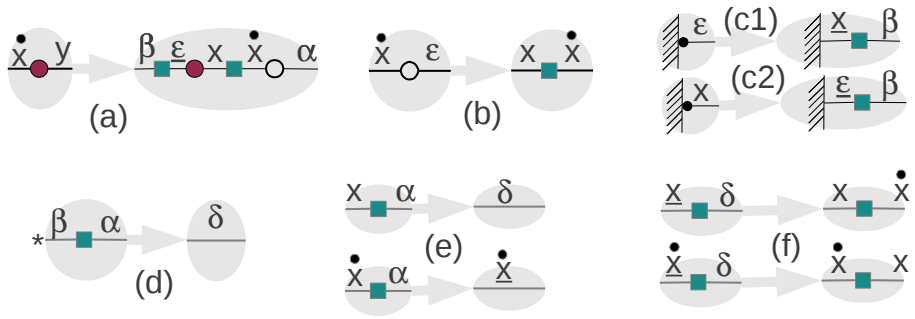

Figure 9. Simulation of the directed buffer. (a,b,c) compiled rule for a node agent including root agent, data agent, host , idle agent. (d,e,f) edge-agent's fixed rule. In rule (d), the symbol '$*$' denotes a collective binding of possibly zero neighbors

Setp2 in $\phi(S)$: Edge-agents are called *slaves* because they execute a fixed rule shown in fig. 9 (d,e,f). They receive orders from the master who owns the edge. which let them replace themselves by a link. An order is encoded as a link labels : β means "let you simplify by pairing with an α " and α means "let you simplify with either β or a link label" (rule (d,e). The effect of these simplification is to restore one edge-agent per connections. The link replacing the edge-agent is labeled δ resp. $\underline{l}$ whenever its final label is not known yet (resp. known to be l). The owner of a link can create arbitrary many connection to the connected neighbor or none at all. As a result, the edge-agent of rule (d), can have arbitrary many β connection or zero.

Step 3 in $\phi(S)$: With rule (f), δ is replaced by the complement of the other edge agent's link label. where complement(l) = $\dot{l}$, and complement($\dot{l}$)= l. The effect is to restore the encoding of the orientation. The role of underlined label $\underline{l}, \underline{\dot{l}}$ is to prevent a node-agent firing before all its edge-agents are done. In other words, it ensures the neighbor-exclusive execution.

7. The label l is dotted, if ownership is kept, which is never the case for the buffer.

The simulation of one complete step needs that all the node-agents update, including those representing idle agents [8]. This is done by leting idle agents apply the idle rule shown in fig. 10 (a). The compilation of the idle rule is shown in fig. 10 (b). It needs the simulation of collective binding, which has not been addressed yet. Assume we want to create connection carrying label x to all y-neighbors. The node-agent insert an edge-agent with link label (x ,χ) is inserted, where x will be doted if created link is owned. That edge-agent does an iterative processing that will create one edge-agents, one for each x-neighbor, using rule fig. 10 (c_1) repetitively. When no link labeled $\underline{\dot{x}}$ is left, then, rule (c_2) ends the processing [9]. The combined effect of collective binding and individual binding in rule (c_1) illustrates well their complementarity, and the resulting expressiveness for node rewriting rule : it enables an iterative processing, link by link.

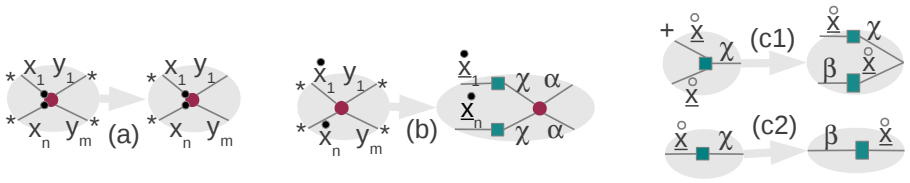

Figure 10. Simulation of Collective binding. (a) the idle rule (b) compilation of the idle rule (c1,c2) edge-agent rule inserting iteratively one edge-agent for each link labeled $\underline{\dot{x}}$. The circle above the labels means that it can be either a dotted, or not dotted label.

4 Yet Higher level Self Developing Network.

Section Outline Our long term goal is to use self development for general purpose programming. In this context, it is relevant to look for enhanced notations of node-rewriting rules that increases expressiveness. The formalisms introduced here are already used in [5]. We first widen the range of neighbors that can be bound using rules called "redirecting rules" which can have a combined transitive effect. We will detail the simulation, and a method to derive a property of confluence. We then propose four different more powerful ways of addressing links. Finally, existing systems can be classified as specific occurrences of SDNs. All these models are programmed on top of directed SDNs, which thus appear as a significant generic improvement upon undirected SDNs.

4.1 Redirecting node-rewriting rule.

We notice that the not-owned (input) connections can be redirected on distinct agents, instead of preserved on the same persistent agent. The neighbor

8. An agent is idle, either because it is not ready, or because it is not in the set A of the considered transition $c_1 \xrightarrow{A} c_2$.

9. Here, it is compulsory that rule c_2 has a lower priority than rule c_1

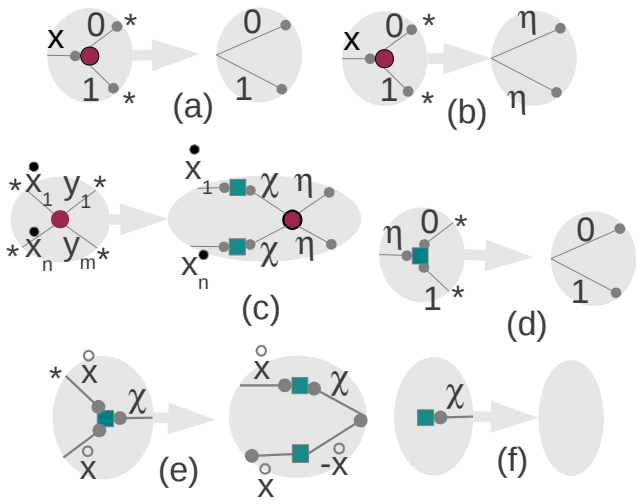

Figure 11. Simulation of redirecting systems. The notation $-l$ permutes dotted and non dotted label, (a) Example of redirecting rule (b) its encoding as a directed rule (c) encoding of the redirected NOP rule (d)slave rule for collapsing the tree (e)(f)slave rule for processing collective binding. $-\dot{l} = l, -l = -\dot{l}$

at the other extremity, which does own the connection, will anyway not perceive any differences, since it sees only the link (its label and orientation are preserved).

Definition 5. *Redirecting node-rewriting rules generalize directed rules : input neighbors can be bound but must be used once in a connecting triplet, with preserved label and orientation.*

Redirecting systems also use a persisting agent, that inherits all the links not bound in the left member. Redirecting systems clearly contain directed systems. In order to distinguish input from output neighbors, the context (and binding) which def. 1 refers to, must now be specified using oriented labels $\overrightarrow{l} \in \overrightarrow{L} = L \times \{0, 1\}$. It includes a label plus a boolean coding the orientation of the matched connection. In def. 5, the constraint on bound input neighbors means precisely 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 connections cannot be redirected towards another input neighbor, since that would change it to an output neighbor and would not respect the constraint on redirection of def. 5. A cycle in a chain of transitive redirections is forbidden, the redirections should form a set of tree, the effect of redirecting is to collapse that tree by transitively gathering all the leaves on the root. An example of transitive redirection is shown fig. 11 (a), and its effect in case of parallel update in fig. 12 (a).

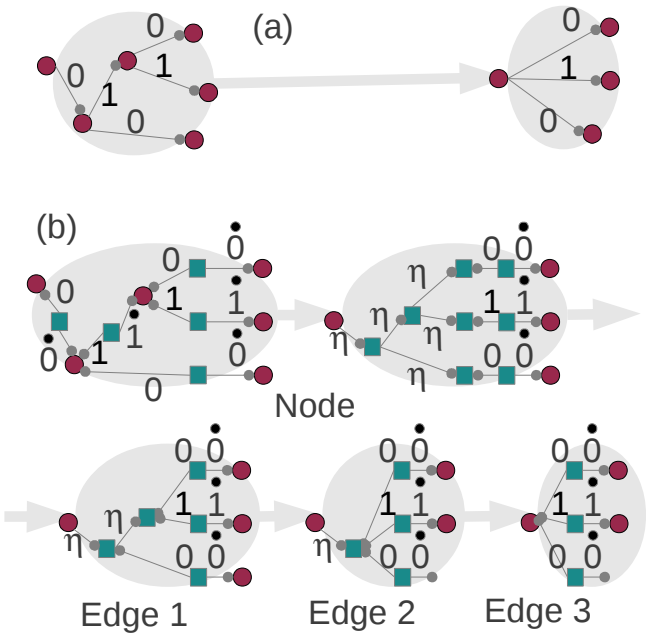

Figure 12. Execution of a Simulation (a) One redirecting rewriting step (b) simulated by 4 directed steps.

Simulation of Redirecting rule.

Theorem 2. *Redirecting node-GRS are high level SDNs.*

Proof : We program redirecting node-GRS using directed node-GRS. So, in a sense, this is "yet a higher level" system. We use a bipartite encoding, with slave edge-agents and master node-agents. The simulation is neighbor-exclusive, since 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 connections, 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 the slave rule fig. 11 (d) which collapses a tree of transitive redirections in a number of steps equal to the depth of the tree.

Since trees can have arbitrary depth, an arbitrary large number of rewriting steps may thus be required for simulating a single step. Redirecting system were originally inspired by membrane systems, for which this number of steps can be made constant : A tree can be represented as a set of encapsulated membranes.

The removal of a membrane is a redirecting rewrite. The removal of several emcapsulated membranes indeed produce a collapse of the tree.

Deterministic Confluent Redirecting SDN Let us study the non-determinism inherent to SDN, in the most general context of redirecting SDN. A first source of non-determinism, valid for rewriting systems 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. A second source of non-determinism happens in 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. A GRS is called deterministic, if the update of a single agent always gives the same configuration, which means those two sources of non-determinism are void.

When considering the updating of many agents instead of a single one, a third 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 in the scheduling. Only confluent rewriting system have an output that is defined independently from a schedule of update. Because we have agents, the simpler concept of commutativity is easier to deal with.

Definition 6. *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. Deterministic elementary SDNs are commutative (theorem 1). Deterministic redirecting systems can be made commutative by adding appropriate delays ; Let us first define a concept of production. 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. The following proposition is used in [6] to prove the confluence of a self developing system called the blob machine.

Proposition 2 (Commutative closure). *A redirecting node-GRS becomes commutative if an agent always waits for removal of input connections that produce labels bound by one of 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 connections created by a_1 through c will all end up on the persisting copy of a_2.

4.2 Refined addressing of links

We now report four ways of more expressively adressing links.

Programmed orientation On top of the orientation of connections used to encode ownership, one sometimes needs to encode a *programmed orientation*. Consider for example a chain of agents such as the one used for the buffer. A programmed orientation can distinguish between left from right. With data-flow agent, it can distinguish boolean input from outputs. With trees, it can identify the father from the child trees. A programmed orientation can be simulated easily on a directed system, using two "opposite" label : for example L and R for right and left. The label is systematically negated when ownership is flipped ; right becomes left and vice-versa. With this agreement, when the connection is owned (resp. not owned), the programmed orientation reads directly (resp. the opposite value must be taken).

Labeling of connection extremities Such labels can be read, and modified, only by the agent at the corresponding extremity. They are useful as a local memory per connections, and makes it easier for an agent to manage its connections without interfering with the neighbors. For example, an extremity index can be used to locally number the connections. Local labels can be incorporated directly in the simulation of directed systems by undirected system. They will be encoded by a distinct label, for each of the two edges of each edge-agents. If a connection is duplicated from one extremity, the extremity labels at the other extremity gets duplicated together with the connection.

Mixed orientation In mixed graphs, connections 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 connection, but using the just mentioned labeling of connection extremities : an extremity boolean flag independently labels each of the two extremities of a connection, by 0, or 1. The possible values for the pair of extremities of one connection are $(0,1),(1,0),(0,0)$. In the case of $(0,0)$, the connection is not oriented, it is *owned* by neither of its two attached agents, which cannot modify the connection, except for a very specific modification which consist in acquiring ownership by setting the ownership flag. A connection cannot be owned at both extremity, thus $(1,1)$ is forbidden. If, by chance, ownership is requested simultaneously at both extremities, a random choice is made. This features a "system" way of breaking symmetries in a SDN.

Ordered list of links Our hardware target for simulation of self development is a 2D computing medium. We restrict networks to be planar in order to simplify the implementation. [10] The hardware support of a node or of a link is a

10. In order to allow the crossing of connections one can explicitly encode crossroads as specific compute nodes.

simply connected set of processing elements. Assume the support of a node is disc-shaped, the links connecting to the disc can be ordered depending on the particular point of contact on the disc. This naturally occurring ordering can be exploited to improve the expressiveness of a node-rewriting rule : in the rule notation, the particular point of contact between a link and the disc representing an agent will be considered informative. This allows to specify rules that can establish one to one wiring between two ordered lists of neighbors, as shown in fig 13. Such a wiring is oft n en needed, if we want to plug together different circuits. As an illustrating example, in the parity network of fig 3, we can avoid the dynamic creation of ports. One simply inserts an agent between the ports and the agent developing the parity. When the parity is developed, that agent does the one to one wiring between the ports and the inputs of the parity circuit.

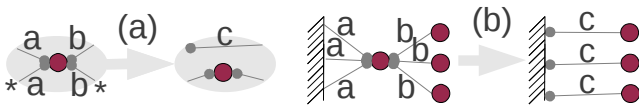

Figure 13. One to one wiring between two ordered lists of connections (a) the rule, (b) execution.

Fig. 14 shows how to simulate the creation of a link, within this enhanced link organization. Each link is represented by two extremity edge-agents. The extremities of a given node are organized as a ring, around the node. Generating a link between two given neighbors can be done only if those neighbors are consecutive with respect to the link ordering, this will also ensure conservation of the planarity of the network. The generation involves a duplication of the link to each of the two neighbors, followed by a merging of edge-agents.

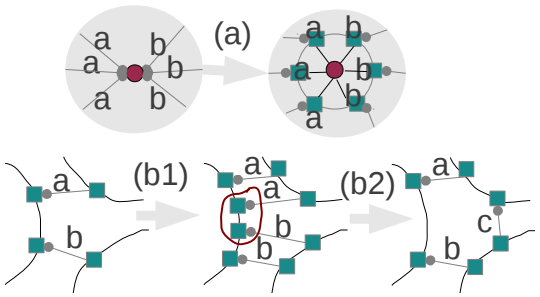

Figure 14. Managing ordered list of links (a) encoding of the network (b) generating a link between two neighbors (b1) duplication (b2) merging.

4.3 Classification of existing SDNs.

Well known systems can be also programmeed on top of directed node-GRS, and classified as specific type of SDNs.

SDN with ever-growing Network The EdNCE graph grammar introduced by Engelfriet and Rozenberg [13] 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.

SDN with acyclic network L-systems are grammars introduced by Lindemeyer [9] 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 can use edge-agents to synchronize the rewriting of all nodes. In the general case, context sensitive L-systems still rewrite agent-wise, but check the state of neighboring agents before applying a rule. For example, this can model a flow of nutrients. The simulation by SDNs must use an intermediate step of communication so that each agent reconstructs 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 the immediate neighborhood, several such communication steps must be composed.

SDN with a constant number of agents Self assembly system focus on maintaining a specific subset of connections 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 [8] 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[12].

SDN with a fixed network. If the rules neither creates nor deletes agents or connections, the network is preserved. Such a degenerated SDN is called "'*static*"' and models a fixed network of finite state automata. *Automata networks* have not been studied very much for their own sake. This is perhaps due

to the difficulty of defining a state-transition for arbitrary neighborhood. Two specific restrictions make it easier to do, and lead to widely studied models :

1. If the next-state transition function makes a commutative, associative reduction of the inputs, and then apply a threshold function, this leads to Artificial Neural Networks (ANNs)
2. If the neighborhood is the same for every automaton, this leads to Cellular Automata (CA).

In both ANNs, and CAs, an agent can directly read the neighbor's state. A simulation with SDNs needs to copy the agent's state on the edge label, and use two edges to allow bi-directional communication. Static SDN can hardly be called self-developing, since nothing is developed. However, modifying the link 's orientation and label already gives an interesting expressive power. Moreover, static SDN can model real hardware. The categorization as static SDNs allows to reuse the same vocabulary, principles and methods for simulating SDN on real parallel hardware.

5 Conclusion, and second half of this work

This work defines "Self Developing Network" (SDNs) as the simplest kind of graph rewriting rule that can be applied in a distributed way. Nodes of the graph are identified as active agents doing the rewriting, and we explore the most generic way in which they can do it. Two different ways of partitionning the network are introduced ; the first, assumes a neighbor-exclusive execution, the second relies on a link orientation. We show that the second kind of system can be seen as just a programming layer above the first. Using link orientation adds expressivness. We use it to program additional layers, in order to obtain yet more expressive ways of specifying distributed rewriting ; one can also implement exisisting well known system which can hence be classified as restricted form of self-development.

The second half of this work [5] shows how to program SDNs using Finite State Automata, resulting in more programmability. FSAs are used to prove an "efficient" intrinsic universality result : it needs only linear time, while a simulation by a Turing Machine would need exponential time. This final results formalize the specific parallel power of SDN, and motivates SDNs as a worthwhile fine-grain formalism for exposing parallelism and locality. As a side effect, the proof also shows that the first kind of system can be programmed with the sedond, thus showing. that our definition does not depend on how we choose to partition.

Références

1. Andrew Adamatzky. *Identification of cellular automata.* Taylor and Francis, 1994.
2. G. Agha. *Actors : a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

3. J. P. Banatre and Daniel Le Metayer. A new computational model and its discipline of programming. Technical Report RR-0566, Inria, 1986.
4. L. Cardelli. Brane calculi. interactions of biological membranes. In *Computational Methods in Systems Biology*, pages 605–614. Springer, 2004.
5. F.Gruau. Self developing networks, part 2 : Universal machines. Technical Report 1550, LRI, 2012. http ://www.lri.fr/~bibli/Rapports-internes/2012/RR1550.pdf.
6. F. Gruau, C. Eisenbeis, and L. Maignan. The foundation of self-developing blob machines for spatial computing. *physica D :Nonlinear Phenomena*, 237, 2008.
7. C. Hewitt, P.Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
8. E. Klavins. Programmable self-assembly. *Control Systems Magazine*, 24(4) :43–56, August 2007. See the COVER !
9. A. Lindenmayer. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *jtb*, 18 :280–299, 1968.
10. Stephen Michael Majercik. *Structurally Dynamic Cellular Automata*. PhD thesis, University of Southern Maine, 1994.
11. Robin Milner. *Communicating and Mobile Systems : the Pi-Calculus*. Cambridge University Press, 1999.
12. 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.
13. Grzegorz Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1,2,3. WSP, 1997.
14. Tomita, Murata, Kamimura, and Kurokawa. Self-description for construction and execution in graph rewriting automata. In *European Conference on (Advances in) Artificial Life, LNCS*, volume 8, 2005.