RAPPORT DE RECHERCHE

L R I

SELF DEVELOPING NETWORK 2 : INTRINSIC UNIVERSAL MACHINES

GRUAU F

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

01/2012

Rapport de Recherche N° 1550

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

Self Developing Network 2 : Intrinsic Universal Machines

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 Le modèle de calcul séquentiel de Turing utilise une source non bornée de mémoire avec un seul agent de calcul. Les ≪ Self Developing Network ≫ (SDN) proposent de considérer une source non bornée d' agents à mémoire bornée. Les agent sont connectés dans un graphe dont les arrêtes sont étiquetées. Ils exécutent des règles de réécriture qui créent d'autres agents et connections. Un agent ancêtre peut ainsi développer un réseau d'agents arbitrairement grand. Ce travail montre comment la programmation des agents est facilitée par l'utilisation d'un automate d'états finis avec actions en sortie (machine de Mealy). Ces actions sont des règles de réécriture et définissent le jeu d'instructions d'une ≪ Self Developing Machine ≫ (SDM). Nous présentons un exemple de SDM appelé la graphe-machine qui utilise 10 instructions, et prouvons que l'expressivité est conservée : Pour tout SDN il existe une graphe-machine qui le simule. A partir de ce résultat, nous prouvons un résultat d'universalité intrinsèque en temps linéaire, alors qu'une simulation via une machine de Turing prends un temps exponentiel. Ce résultat met en perspective le parallélisme présent dans l'auto-développement.

The Turing sequential model of computation uses an unbounded source of memory, and a single Processing Agent. Self Developing Network (SDN) propose to consider an unbounded source of agents having bounded memory. This PEs are wired in a graph with labeled links. They execute node rewriting rules which create other agents and links. An ancestor agent can thus develop an arbitrary large network of agents. This work shows how to program agents using a Finite State Automaton with output actions (Mealy machine). The actions are rewriting rules, that define the instructions of a Self Developing Machine (SDM). We present an example of SDM called the graph-machine using ten instructions, and prove that the model expressiveness is preserved : for any SDN, there exists a graph machine that simulates it. Based on this result, we prove an intrinsic universality result in linear time, while a simulation by a Turing Machine needs exponential time. This results put into perspective the parallelism present in self-development.

Keywords: Turing, Self development, Self developing machine, model of computation, massive parallelism, universality, intrinsic universality

1 Motivation and review

In [3] Self Developing Networks of agents (SDN) are introduced as node replacement Graph Rewriting Systems (node-GRS) which rules can be executed in a decentralized distributed way by the nodes, considered as agents. The goal of this work is to introduce the use of Finite State Automata (FSA) to turn this formal model into a programming model, and ground it with an intrinsic universal property with linear time complexity. We first contrast our approach with other existing programming models also based on dynamic creation of agents.

Multi-threading : Generating a new agent looks very much like forking a thread. The difference is in the grain size : SDN agents have finite state, the memory size needed for each agent is known statically. When forking occurs, only a fixed size structure needs to be allocated, instead of a data segment intended to store a large stack. In a real implementation, generating a new agent is a "normal" instruction which cost must be comparable to an arithmetic machine instruction. In contrast, a thread should be sufficiently coarse grain to keep a processor busy for thousands of clock cycle, independently from the other threads. This is because the CPU time cost of switching between thread contexts is large compared to the cost of performing one operation, and it has to be amortized. Because of its finite state, an agent cannot compute anything meaningful by itself. It is only by unfolding a circuit and communicating data along its edges that a non-trivial computation can take place. In short, multi-threading is mainly a method to expose parallelism by outlining chunks of computation that can be carried out independently, while SDNs provide circuits with a dynamic architecture.

Cooperative computing : Cooperative computing [6,5] does consider a collection of FSA that can exchange messages, and create new FSA. The framework is synchronous : at each time step, automata receive messages, change their inner state, and send new messages ; The network used for communication is not explicitly instantiated at run time. Instead, a *global name space* is used to address FSA : any FSA can communicate with any other, using its name or id, this implicitly requires a shared memory, which is not conducive to scalable parallel computing. In self development, an agent can communicate with another, only if the links is explicitly present. The network along which communication will happen is exposed and can be dynamically mapped on hardware.

Structurally Dynamic Cellular Automata (CA) [1,4] : They 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, they are all

Chemical inspired systems : In chemically inspired model, agents are molecules that interact pairwise with each other and generate new molecules. Such model have been used in [2] to maximize the parallelism in the description. The real distributed execution of such models is not obvious 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.

Static instruction.		Dynamic instruction	
SELECT x	Selects input links labeled x	MV	Redirect selected links
LABEL x	Sets the label of selection to x		towards the leader
EMPTY	Tests if selection is empty	RM	Removes selected links
FLIP	Flips orientation of selection	CP	Duplicates selected links
WAIT	Waits removal of input links	NEW	Creates a new agent
ELECT	Elects a leader within selection		

Table 1. The ten instructions of the Graph Machine.

2 Self Developing Machines (SDM)

In [3] we present the formal foundation of Self Developing Networks (SDN) as a sub class of Graph Rewriting Systems (node-GRS), where single nodes are replaced, and the node rewriting rule can be executed in a decentralized distributed way by the network nodes themselves which are considered as active agents. A rule's left member matches an agent's state $q \in Q$, and also its connections by using a multiset $C : L \mapsto \mathbb{N}$ of labels $l \in L$: let $\dot{C}(l)$ be the number of occurrences of l in C, the agent should have at least $\dot{C}(l)$ connection labeled l, whose corresponding neighbors are individually bound. The rule replaces an agent by n new agents and new connections. The right member lists the states of the new agents, and specifies new connections using triplets : (new label, source, target). The source and target can be either 1- one of the new agent 2- an individually bound neighbor 3- all l-neighbor not individually bound for a given label l. This is called collective binding, and can be done only on one extremity.

Programming directly with rules is not very practical and can be eased as follows : Development really includes two kinds of processing : a static calculation part, involving solely the agent's state, and a dynamic part, which is pure graph rewriting, and can be encapsulated into a fixed, small set of well designed elementary rules. The development can then be programmed by combining these primitives using the classic framework of Finite State Automaton (FSA).

Definition 1 (Self Developing Machine (SDM)). *It is an SDN parametrized by a FSA. The node rewriting rules are the FSA's output actions.*

The output actions are the machine's *instructions*, the FSA is the machine's *program*. Agents update by first computing the FSA new state, and then rewriting according to the FSA action. For each created agent, an instruction must specify an input for the agent's FSA. Created agents inherits the same FSA and copy the same new state, it is that distinct input which differentiates them by letting their next transition lead to a distinct state. A machine is defined by its instruction set, its program, and its initial configuration which includes an ancestor agent starting from the FSA's initial state, and external agents called hosts with connections to the ancestor called ports. We found a nice example of Self Developing Machine (SDM) in [7]. It uses 8 instructions devised so that each agent conserve always exactly three neighbors. Its goal was to simulate the Von-Neuman self reproduction.

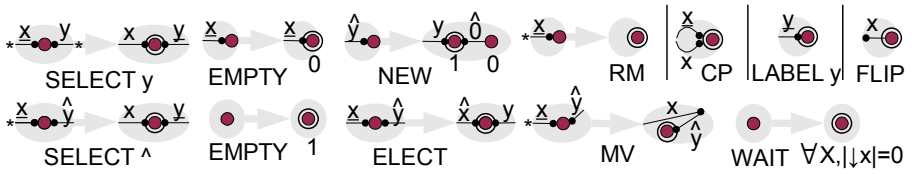

Figure 1. The graph-machine instruction's rule. The input appears below the agent.

3 The graph-machine

We design an instruction set defining an SDM called the graph-machine, its first purpose is to show that the new formalism of SDM preserves intact the expressiveness of SDN as stated by theorem 1. All what can be done with SDNs can be done with the graph-machine. The instructions, listed in table 1 are elementary : they creates agents one by one and process links label by label, by selecting them and then either duplicating, deleting or moving them, or changing their label and orientation. Any graph can be developed node by node, link by link, which explain the appellation "graph-machine". The rules associated to the instructions shown in figure 1 use the intuitive graphical convention described in [3] : The left member locates bound neighbors by placing them around a light gray disc and shows the label of their connections (the symbol '*' indicates collective bound neighbors), the right member reproduces the same disk, and assumes the neighbors conserve the same location on that disc. We use directed node-rewriting rule defined in [3] which rewrite nodes in graphs whose links are oriented, and only output links are considered to be owned, and can be modified. This convention allows two neighbors to update in parallel. Ownership is represented using a tiny black disk at the source of the link extremity. In the right member, a distinguished agent called the persisting agent is designated using an extra circle around it. It keeps all the not owned links, and more generally, all the links omitted in the left member. In [3], it is proved that directed node-GRS can be simulated in a distributed way, thus checking the definition of SDNs.

Static instructions can modify the state, link's label and orientation, or delay the update. Dynamic instructions modify the network itself by creating or deleting links and agents. The links's label includes an integer $x \in < 1 \dots n >$. Adressing links is done in two ways : instructions `SELECT` x select links labeled x, and instruction `ELECT` randomly chooses a leader amongst the selected links. The agents need to remember the current selection (resp. leader) : they use a flag _ (resp ^) encoding selection (resp. leadership) which are encoded as complementary Boolean labels. Selection and leadership have a local scope : whenever a link's ownership is given back (either by move or flip), the link is deselected and unelected. `ELECT` does an elementary individual binding, It is the only non deterministic instruction. The other instructions either do collective binding which is deterministic, or bind the leader which is guaranteed to be unique. The leader can be selected using instruction `SELECT` ^. A link cannot be simultaneously

elected and selected. Insruction `LABEL` x changes the label of previously selected links.

In an SDN, a rule specifies a new state for each created agent, with SDM, it now specifies their next input. Most instructions use a constant input which is useless and thus omitted in the rule description. A Boolean input is provided by instruction `EMPTY` to return the test result, and by instruction `NEW` to distinguish the newly created agent from the persisting agent. The newly created agent is initially connected with a link labeled $\hat{0}$. Instruction `CP` duplicates selected links, the duplicata are deselected.

Fig. 2 shows a graph-machine program for inserting edge-agents (squares) on multiple links labeled x. The executing agent has another link labeled y. This example is chosen because the development involves a loop, and it is used in the proof of theorem 1. The transition diagram is drawn like a flowchart, with diamond shaped box for instruction delivering a Boolean input. The development creates one agent for each connection. After each agent creation, the persisting agent (up) is moving, duplicating and deleting links to the new agent (down) which is waiting to do the next loop iteration. The automaton can be compactly programed as a procedure `one2one`(x, y).

The following theorem shows that graph-machines capture all possible developments. Incidentally, it also proves that basic SDNs, which use undirected networks and a neighbor exclusive rule system can be simulated by directed node-GRS. Since the converse has been proved in [3], both classes are equivalent.

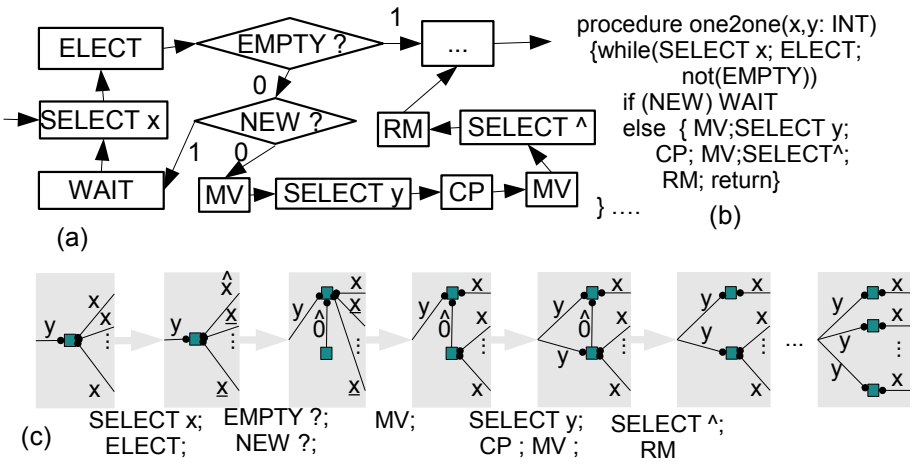

Figure 2. A graph-machine program (a) FSA (b) code (c) development.

Theorem 1. *SDNs are simulated faithfully by graph-machines.*

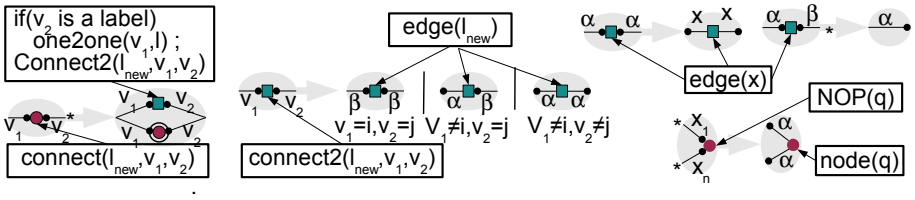

Figure 3. Development generated by procedures for the simulation of undirected SDN.

```
procedure node(q: integer, S: developing rule)
{ WAIT;
  for all rule (q,C --> q1, ..,qn, T)
      if (matched(C)) glue(q1,..,qn,T)
      else clean(C)
  NOP(q) }
procedure NOP(q: integer)
{ for all label l in L
      SELECT l; LABEL alpha
  SELECT alpha; FLIP; }
function matched(C: multiset)-->boolean
{ forall (x,n) in C
     for i = 1 to n
         SELECT x; if(EMPTY?) return FALSE
         ELECT; SELECT LEADER; LABEL (x,i)
  return TRUE }
procedure clean(C: label multiset)
{ forall (x,n) in C
     for i = 1 to n
         SELECT (x,i); LABEL x; }
procedureglue(q1,..,qn: integer,
         T: list of connecting triplets)
{ for i = 1 to n
      { NEW{ node(qi)}
      SELECT LEADER; LABEL i;
      }
  for all triplet (lnew,v1,v2) in T
      connect(lnew,v1,v2)
  for all v in L union C union <1..n>
      SELECT v; RM; }
procedure connect(lnew: label,v1,v2: adress)
{NEW
 {WAIT;
  if (v2 is a label) one2one(v1,v2)
  connect2(lnew,v1,v2)
 }
 for i =1,2 { SELECT vi;CP;MV; }
 SELECT LEADER; RM; }
procedure connect2(lnew: label,v1,v2: adress)
{for i =1,2
     if (vi is an integer) LABEL alpha
     else LABEL beta
 SELECT alpha; FLIP; edge(lnew) }
procedure edge(x: label)
{WAIT;SELECT beta;
 if (EMPTY?) { SELECT alpha; label x; FLIP}
 else { ELECT; SELECT beta; MV;
      SELECT LEADER; RM} }
```

Proof : By the definition given in [3], SDNs are parallel node-GRS that can be simulated by neighbor exclusive (two neighbors are never simultaneously ready) undirected node GRS, that are defined as basic SDNs. By composition of simulations, we only have to simulate those. Consider a basic SDN S and a configuration c. A simulation is defined by a transformation mapping configuration of S to a graphe machine's configuration. We use a bipartite master slave transformation of the kind introduced in [3]. Slave edge-agents (squares) are inserted on each edge and execute the procedure call `edge`(x) where x is the edge label. The two links of each edge-agents get the same label α. node-agents are the masters, they execute `node`(q, S) where q is the agent's state. Procedures `edge` and `node` represent FSA since they need only bounded memory. Their code is shown above, and the corresponding developments in fig. 3. edge-agents isolate node-agents from each other to prevent interference between a node-agent trying to match a rule, while a neighbor is giving him new connections. Edge-agents and node-agents wait to own all their connections before starting. More generally, the execution is owner-all update for every generated agents, and therefore neighbor exclusive.

A simulation step starts with edge-agents which set the label of their two links to x and flip orientation to give ownership to node-agents. Then, node-agents try to match each rule, one by one. The order of trial should be in decreasing priority, with a random re-ordering of rule having equal priority (to implement a fair execution). The Boolean function `matched`(C) checks that the link count for each label corresponds to the specified multiset of label C. At the same time, it also does the individual bindings by relabeling the i^{th} counted $l-$link with label (l, i) instead of l. The only place where non determinism appears is in this matching-binding phase, therefore the simulation is faithfull as defined in [3] : any execution in the simulating system correspond to an execution in the simulated system. If the matching fails, the procedure `clean` removes those indexed label. If it succeeds, the procedure `glue` creates new node-agents, linking them with their index $i \in < 1 \dots n >$ as label. It then process the connection triplets (l_{new}, v_1, v_2) specifying new connections.

At this point, a node-agent has set the labels of its links to exactly represent the possible source and target $v_{i,i\in\{1,2\}}$ between which to draw new connection : $v_{i,i\in\{1,2\}}$ can be either the index $i \in < 1 \dots n >$ of a created agent, an indexed label $(l, i) \in L \times \mathbb{N}$ referring to an individually bound l-neighbor, or a label $l \in L$ for collective binding of all l-neighbours not individually bound. Consider first individual binding : the processing of each triplet (l_{new}, v_1, v_2) creates a new edge agent and its two links connected to each target v_1 and v_2. Those two links are labeled α (resp. β) and are owned (resp. not owned) if v_i was an integer, (resp. an indexed label). The β links reach an edge-agent already present before the simulation step, this agent will move those β links towards its unique α connection (unique because of neighbor exclusion of the simulated system). For collective binding, a call to the already defined procedure `one2one`$(v1, l)$ (see fig. 2) is inserted to generate as many new edge-agents as needed for each bound.

4 Efficient Intrinsic Universality

We now take into account the time and space complexityof development. If the the number of links that an agent can ever get is upper bounded, it becomes reasonable to assume that agents update in constant time, since everything is bounded, (states, label, links). For such system called *bounded degree SDN*. the size of a configuration is the number of agents, up to a constant factor which is omitted if we study complexity.

Definition 2. *The free time (resp. space) complexity of a derivation in a bounded degree SDN is the minimum number of step (resp. the max size of generated configurations).*

Time is measured for the quickest equivalent derivation where ready agents do not wait, but space considers the max size of all the networks that can be generated with different schedule. The time and space complexity of a simulation using a transformation ϕ between S and S''s configuration is $O(f(t))$ (resp. $O(g(s))$) if there exists a constant K such that for any derivation x in

$S, \text{Time}(\phi(x)) < K * f(\text{Time}(x))$ (resp. $\text{space}(\phi(x)) < K * g(\text{space}(x))$). If f and g are the identity function, it means that a derivation step is simulated in constant time, and the support of each agent is finite. The simulation is called *linear* in space and time.

Consider a toy development where each agent duplicates at each time step. An execution on a Turing Machine, is forced to iterate through all the agents to simulate one parallel derivation step, therefore the simulation time for one step increases exponentially as development unfolds. In contrast, it is possible to provide a universal SDM that needs only constant time. Note that since everything is bounded, the only thing left that makes the free complexity unrealisable is the finite number of dimensions of our physical space, which cannot accommodate an exponential growth in linear time.

Theorem 2 (Efficient Intrinsic Universality). *There exists a bounded graph-machine that simulates faithfully all bounded SDNs in linear time and space.*

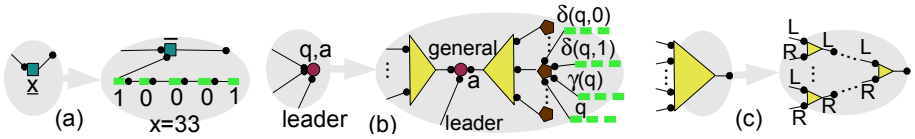

Figure 4. Bipartite transformation used for intrinsic universality. Labels are omitted for clarity, except for (L)eft and (R)ight. (a) edge-agents (rectangle) use bit-agents (squares) (b) branch-agents (triangle) (c) the general owned), right : (disc) with state agents (pentagones)

Proof : We have to simulate only owner-all update graph machines, since only those machines where used in theorem 1. Consider such a graph machine M and a configuration c. We use the bipartite master-slave transformation shown in fig. 4, which needs several stages of decomposition. Edge-agents uses a chain of bit agents providing a binary encoding of their labels and a selection-flag storing if the link is currently selected. The master node called "general" connects to the leader edge-agent (down) plus two binary tree of branch-agents encoding a multiset of edge-agents (left), and a multiset of state-agent (right). Information encoded as trains of bits travels back and forth between the edge agents and the state agents, through the general and orchestrated by the general. Bit-agents and branch-agents are generic library elements. A chain of bit-agents can receive a new value, duplicate, and test for equality. Branch-agents simulate arbitrary large degree using a fixed degree of 3 : one link to the father and two links for each of the two child tree labeled L and R. A tree of branch-agents can duplicate, propagate a trains of bits downwards or upwards, extract a sub-tree, move one distinguished leave upwards, and make a logical `OR` of value comming from the leaves. A programmed orientation [3] allows to distinguish the two children from the father. For example, it explains why x and y are distinguishable

in fig. 5 (a). Any binary tree is a possible encoding, therefore the transformation is non deterministic, it describes a set of acceptable representations. A branch-agent can be in two modes : upward and downward. Downward branches own their two child links, but not the father link, they listen to commands sends from the root and propagate them downwards to the leaves. Upward branches own their father link, but not all their child links. At the beginning of a simulation step, if all the links are owned (output), all branches are downward which is the simplest case represented in fig. 4 (c). However, is some links are not owned, they connect via upward branches, and all the branches on the path leading to the root are also upward. The upward branch's program shown in fig. 5 (a) waits that all the child links are owned, (resp. or removed) to switch to downward mode (resp. or to simplify). This action triggers the father to also become downward, and later the father of its father, until the root which is the general. At this point, the whole edge-agent tree contains only downward branches. The general knows that it owns all its links, and can update. The instruction `WAIT` needs not be considered because the simulated machine is owner-all update.

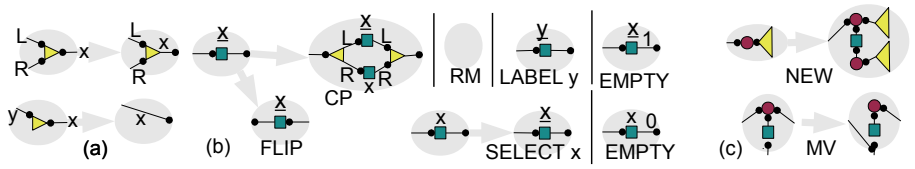

Figure 5. Development for intrinsic universality. (a) Upward branch node (b) Edge-agent listening to the right (c) General executing `NEW` and `MV`.

The state-agents tree represents the automaton. At the beginning of a simulation step its branches are all downward. Each state-agents represent a state q of the automaton, it links to four sequence of bit-agents, coding q the next states $\delta(q,0), \delta(q,1)$ for the two possible Boolean inputs and the action $\gamma(q)$. A simulation step starts by computing a transition in the automaton. The general propagates the FSA input a downwards to the state-agents. The one state q that is currently activated, responds by propagating upwards the next state $\delta(q,a)$, which is then forwarded downward to all the states, for equality test. The state $q' = \delta(q,a)$ becomes activated, and responds by propagating its action $\gamma(q')$ upward to the general. The codop of the instruction is received first, and triggers a distinct processing. The action `SELECT` x (resp. `LABEL` x, `CP` , `RM`, `EMPTY`, `FLIP`) is broadcasted through the edge-agent tree downward to the edge-agents which then matches the bit sequence and sets the selection-flag accordingly, (resp. writes a new label, generates a copy of itself, deletes itself, propagates the ownership flag, change the side for listening to orders) as shown in fig 5 (b) Instruction `EMPTY` and `FLIP` also have an effect on the traversed tree branches, `EMPTY` let the branch do a logical OR, and `FLIP` leaves the branch in upward mode.

The other instructions needs more processing from the general illustrated in fig.5 (c). Instruction `NEW` duplicates the entire tree of state-agents, and creates a new general. Instruction `ELECT` does two downward-upward traversal : the first traversal flag branch nodes which contains a selected link amongst their leaves. The second traversal draws a random path to one of them by flipping a coin when it needs to choose between Left and Right sub-tree, and then moves back the elected edge-agent along that path, upward to the general. Instruction `MV` involves first a downward-upward phase on the edge-agent tree to extract a sub-tree of all selected links, and this sub-tree is moved through the leader connection.

The universal automaton also needs to develop the first instantiation of state-agent tree representing the particular automaton that will be executed. This tree is loaded in parallel from the ports, by distributing the states into as many groups as ports. In the end, we obtain a universal automaton U that combines all the slave's automaton (branch-agent, bit-agent, state-agent, and edge-agent), plus the general automaton, and the loading-automaton. An arbitrary SDN is simulated by U by first simulating it with a basic SDN (definition) encoding it as a graph-machine automaton (theorem 1), and finally loading that automaton from the host. The degree of U is bounded by 5, and it uses a fixed set of labels. The simulation works for unbounded SND, but for bounded SDN, it is linear in time and space, since the number of branch nodes, and bits per label will also be bounded.

5 Conclusion

In this work, we ease the programming of Self Developing Networks (SDNs), by using Finite State Automata (FSA), and expressing the development using instructions defining a Self Developing Machine. We first present a 10-instruction machine called the "graph-machine" and prove that for any SDN, there exists a graph-machine that simulates it. We then construct a particular graph machine's FSA U that can simulate all the graph-machines used in the first result. Furthermore, if the degree of the simulated SDN is bounded, this intrinsic universal simulation needs only linear time and space ; In contrast, a sequential Turing machine needs a time which can grow exponentially. This result thus highlights the superior parallel processing power of SDNs brought by considering an unbounded source of Processing Elements.

The FSA U needs only 8 of the 10 graph-machine's instructions, showing that two of them are not of essential nature. Instruction `WAIT` is discarded by restriction to owner-all update. Election amounts to choosing a leave in a tree, and is simulated by establishing a random path through a binary tree to a leave, without using instruction `ELECT`. The remaining 8-instruction are all deterministic, non determinism of the simulated SDN is simulated purely by non deterministic in the FSA U (flipping coins).

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

Références

1. Andrew Adamatzky. *Identification of cellular automata.* Taylor and Francis, 1994.
2. J. P. Banatre and Daniel Le Metayer. A new computational model and its discipline of programming. Technical Report RR-0566, Inria, 1986.
3. Frederic Gruau. Self developing networks, part 1 : the formal system. Technical report, INRIA, 2011. submited to the Turing Centenary Conference, 2012.
4. Stephen Michael Majercik. *Structurally Dynamic Cellular Automata.* PhD thesis, University of Southern Maine, 1994.
5. L. Mandel and M. Pouzet. ReactiveML : a reactive extension to ML. In *PPDP '05*, pages 82–93, New York, NY, USA, 2005. ACM.
6. M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *PPDP '04*, pages 203–214, New York, NY, USA, 2004. ACM.
7. 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.