

A network computational model's for Software Defined Networking

Andrés Laurito¹ and Hernán Melgratti²

Departamento de Computación - FCEN -UBA
andy.laurito@gmail.com

Abstract. In this paper we define a computational model for network's programming languages. In this paper we define a computational model for network's programming languages that use Software Defined Network (SDN) paradigm. For this purpose, that use Software Defined Network (SDN) paradigm. For this purpose, network's will be model as graphs, and a graph grammar semantic will be defined. network's will be model as graphs, and a graph grammar semantic will be defined. Production's in the grammar will model actions that can take place in the network, and programs Production's in the grammar will model actions that can take place in the network, and programs will be just a sequence of instruction's modifying a starting graph.

will be just a sequence of instruction's modifying a starting graph.

After graph grammar's definition, we will extend this grammar to a type graph grammar, and we After graph grammar's definition, we will extend this grammar to a type graph grammar, and we will explain how this new grammar can be applied to create a type programming language will explain how this new grammar can be applied to create a type programming language for networks. Finally we will show how this computational model is well defined, by using it to for networks. Finally we will show how this computational model is well defined, by using it to create a compiler for a new network programming language recently emerged called Nemo.

1 Introduction

Nowadays networks have become a crucial resource of any software application. Technologies for improving code crafting are emerging constantly. By contrast, network's have several issues to be solved. Some of this issues may be tagged as

- **Debugging problem's:** It is rather difficult to debug problems in network's using most known tools such as ping, traceroute, netstat, between others.
- **Changeability problem's:** changing traffic rule's, or security rule's in network may imply modifying hundred, or even thousands code line's distributed in several devices around the network.
- **Deployment problem's:** most changes are made manually in several devices, making human mistakes easier.

Is it possible to do better ?, Can software be of help in this task?, Is it possible to know when network change's will break something before applying them?.

We believe that all this question's have possitive answers, and the best way of solving this problem nowadays is using network's programming languages which are build over the Software Defined Networking (SDN) paradigm.

Multiple SDN's programming languages have emerged over the past year's (some examples of these are Pyretic, Frenetic, Nemo, etc.), having each of them different approaches to solve some of the problems described before. Some of the principal lack's of the network programming languages discipline nowadays are formal grammar definitions, verification, validation, testing, software metrics and quality assurance, between multiple others.

Missing this properties makes harder to build some tools for network programming languages, such as compiler's, debugger's, coding enviornment's, etc. This limitation's makes the network operator's job difficult, even when using some of this programming languages.

The main contribution of this paper is presenting, as far as the authors know, the first network computational model based on the definition of a graph grammar. In section 2 we will define the problem to be solved with several examples and assumptions. In section 3 we will define the graph grammar, and we will extend this grammar to a type graph grammar in section 4. In section 5 we will use this grammar to show how we can solve the problems specified in section 2, and in section 6 we will check how the grammar can be used for building a simple compiler for the Nemo programming language. Finally, in section 7 we will give our conclusion and in section 8 we will introduce further work to be done.

2 Definning the problem

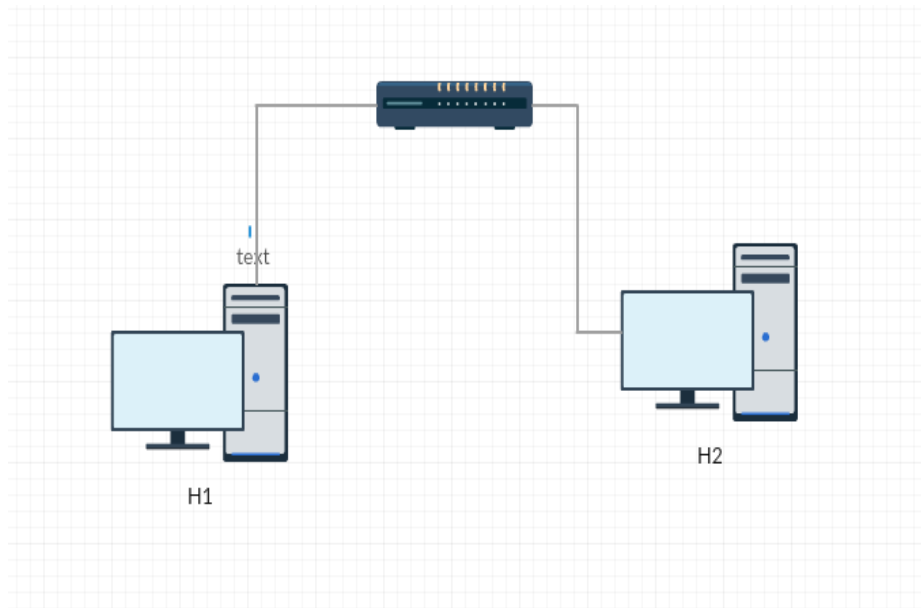
Let us remember that a program may be defined as a sequence of instruction's that executed perform a task in a computer or multiple computer's. Every program can be model as a Turing's machine or a finite lambda calculus computation, and despite of the model chosen, the operations are performed over a structure (usually this structure is a memory). In consequence, every network program is model as a finite sequence of operations over a structure.

Our first proposal is to define this structure as an oriented graph, since network's can be easily mapped to graphs, and also is perfect reasonably to think program's operation as graph's rewriting. After this reasoning, the second propose is to define a graph grammar in order to model all the network program's that may be written in any netowrk programming language.

In order to define the grammar production's, we will narrow to model programs that solve the following problems in networks where each device support's SDN:

First Problem:

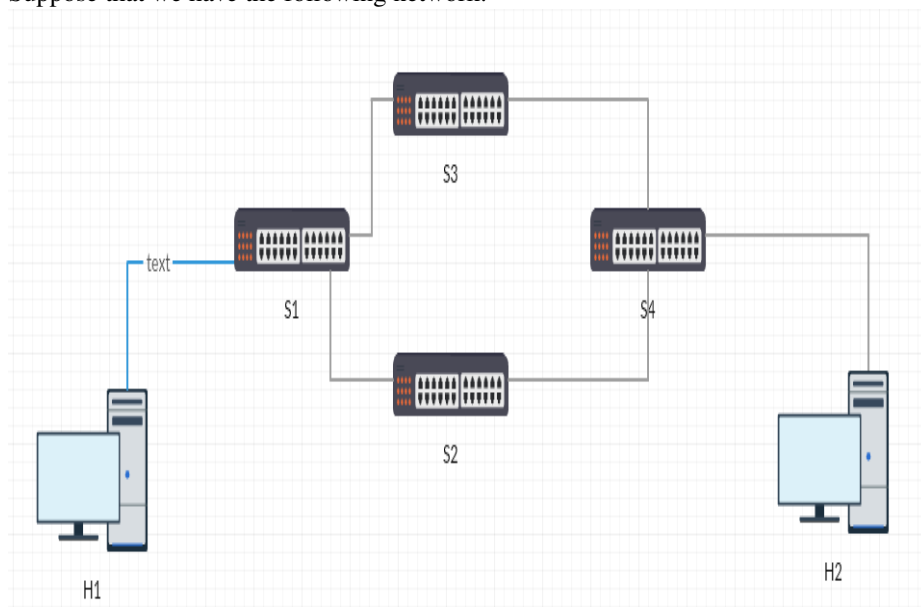
Suppose that we have the following network:



where H1 want's to establish an ssh connection with H2, and let's assume that there are not forwarding rules in order to establish such connection's, and also H2 has not an ssh server client to fulfilled this connection.

Second Problem:

Suppose that we have the following network:



where H1 wants to establish a ssh connection with H2, using S1- \rightarrow S2- \rightarrow S4 path, and let's

assume that there are not forwarding rules from S2 to S4.

Defining a graph grammar allows us to check easily some of the problems described before, particularly static properties, such as examples 1 and 2 (as we will see in following sections). However, dynamic properties (such as applying some load balancing policy when a queue starts to drop more than 50% of packets) are not easily checked by grammar production's. Some problems required to execute our program in a traffic network, however our approach is identifying problems in program's before they actually happen in the real network. In order to identify this class of problems, we propose an execution of some of the grammar production's in a simulator which will be inserted inside the language compiler. The program will then be executed in several simulation's, in order to check if the production applied to the actual graph is possible. For doing this, we will mapped the dynamic grammar production's to devs (a formal language specification for defining simulator's), and we will run a simulation using this formal specification.

Having defined the problem that we will be solving, let's define the primitive operations that we will used in our computational model.

3 Graph grammar production's

We start by defining the most basic grammar production's. These productions are the one's that allows us to modify the network topology:

- Creating a node

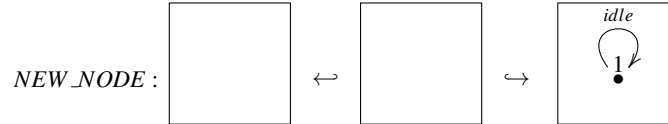


Fig. 1. Creating a node

This production creates a new logical node (this means that the node could be either physic (which mean's it's a physical host with an associated address) or logic (this mean's a group of physical nodes, meaning that the address could be a CIDR).

It is important to remark that the created node is in an idle state, meaning that is not interacting with any other node in the network.

- Deleting a node

REMOVE_NODE :

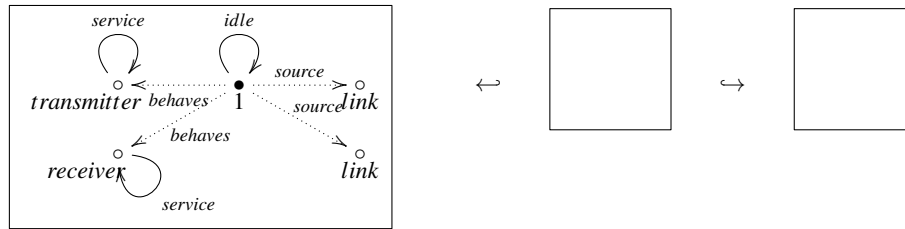


Fig. 2. Deleting a node

This is the production that will allow us to delete node's from our network.

In order to apply this condition, we must satisfied that this node is neither linked against a link, nor linked against a service.

QUESTION: IS THE NEGATIVE CONDITION SAYING THAT ALL OF THIS PROPERTIES MUST BE PRESENT TO NOT FULLFILLED THIS RULE?, OR IM SAYING THAT IF ONE OF THEM IS PRESENT, THEN IT FAILS (this is what I want).

– Node properties creation

NODE_PROPERTY_CREATION :

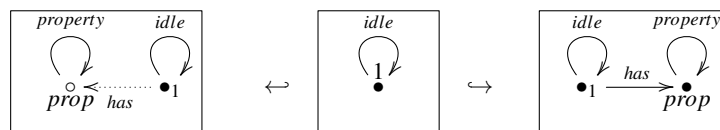
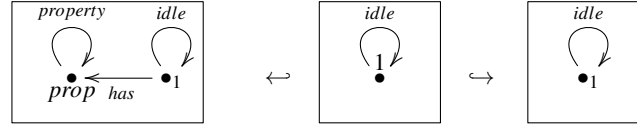


Fig. 3. Node properties creation

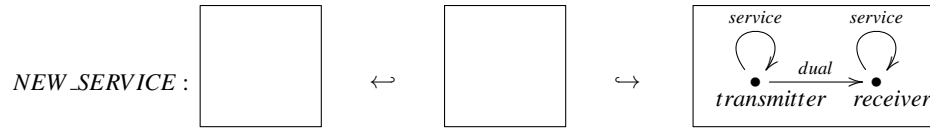
This production allows us to create a node's property. The only condition to be satisfied to apply this production, is that the property to be created can't be an already defined property for node 1.

– Node properties deletion

NODE_PROPERTY_DELETION :**Fig. 4.** Node properties deletion

This production allows us to remove a node's property. At first, I see no need to impose some restriction to applied this rule.

– Service Creation

**Fig. 5.** Service Creation

In this production we define a new service. A service is either some property implemented in nodes, or implemented in another node, which will be used for generating flows, this means that, when a flow is created, this flow will be of type service. The fact that the main idea is to send and receive information, is represented as dual nodes representing a transmitter and a receiver in the graph.

– Service Deletion

REMOVE_SERVICE :

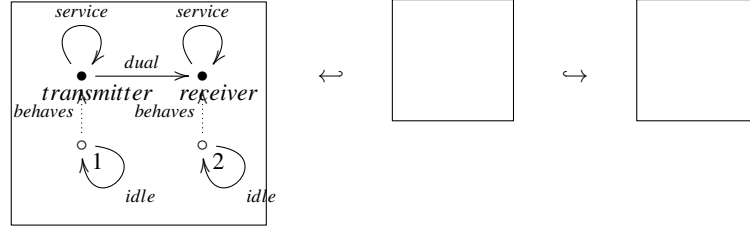


Fig. 6. Service Creation

In this production we allow to remove a service, whenever the service is not being used by a node (this is represented as a node linked to one of the service endpoint's).

- Node behaves as some endpoint of communication

NODE_IMPLEMENTES_ROLE :

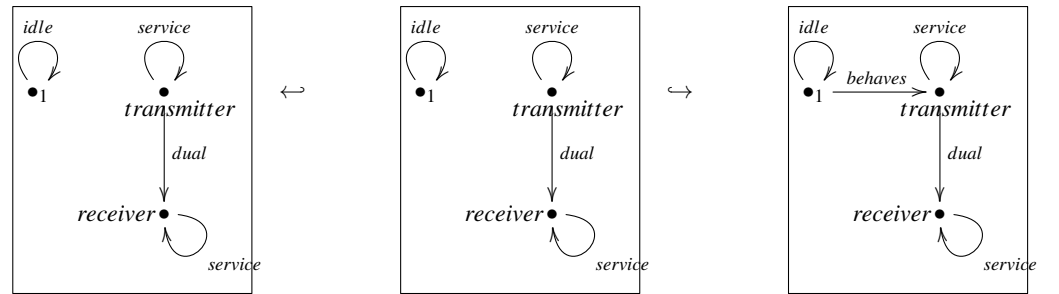
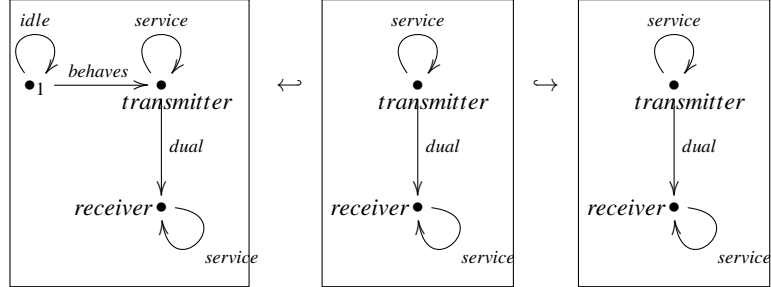


Fig. 7. Node implements role

This production is used to make node's to behave as one of the endpoints in the communication of services. One node could eventually behave as either endpoints (an example of this situation could be a http connection, where the node is both client and server).

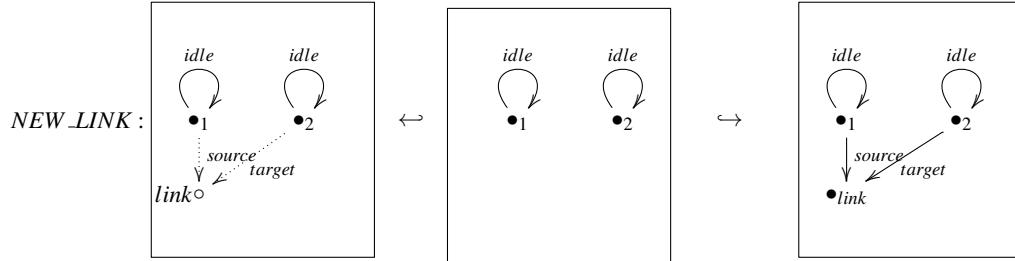
- Node stops behaving as some endpoint in the communication

NODE_STOPS_IMPLEMENTING_ROLE :**Fig. 8.** Node stops implementing role

This production is used to make a node stop behaving as one of the communication's endpoint.

I THINK THIS SHOULD BE THE STANDARD TO USE FOR EVERY PRODUCTION THAT DELETES PROPERTIES OR NODES FROM GRAPH (REMEMBER THAT SERVICES, NODES CAN STOP WORKING WHILE THE NETWORK IS RUNNING).

– Link creation

**Fig. 9.** Link creation

Here we define a new link between nodes 1 and 2. The only condition to apply this production is that both nodes must be in an idle state and neither of them should have an existing link between them. This restriction is denoted by the dot lined in L, wich represents the negative condition.

Note: Being in an idle state does not mean that they are doing nothing!. An idle state behaves like a wildcard, meaning that from this state, multiply actions can be

applied to the node.

– Link deletion

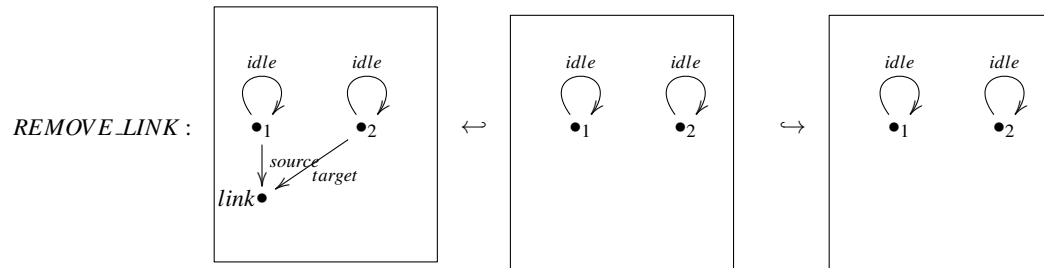


Fig. 10. Link deletion

This is the production used to remove links from nodes. When we allow this production to be applied without constraints, we are trying to represent real states that could happen, since links usually fail without previous warning.

– Link's property creation

NEW_LINK_PROPERTY :

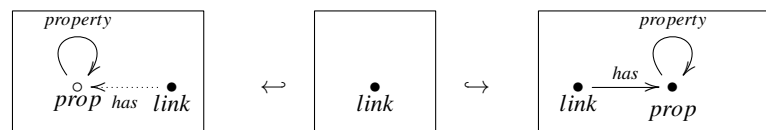


Fig. 11. Link's property creation

As in the node's creation property, this production allows us to create a new link's property. The condition to be satisfied to apply this production, is that link cannot have defined the property to be created. This properties are going to be perhaps physical properties in physical links (for example, we can define here bandwidth, average latency, perhaps if it's either ethernet or wifi), and some other properties in logical nodes.

QUESTIONS:

- 1) What properties could be assigned to a logical link?
- 2) Can I always define properties in a node? (It really doesn't matter what is going on with that link in the network?. Perhaps it happens something similar as nodes,

that we need some 'idle' state, for example, an empty state).

- Link's property deletion

REMOVE_LINK_PROPERTY :

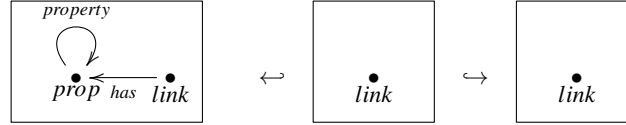


Fig. 12. Link's property deletion

This is the production that allows us to remove properties from a link.

So far we have defined productions that are intended to be use between the interactions with the elements in the network. With the following productions, we will start using the productions listed before, in order to crate interaction between the elements in the network.

- Flow creation

NEW_FLOW :

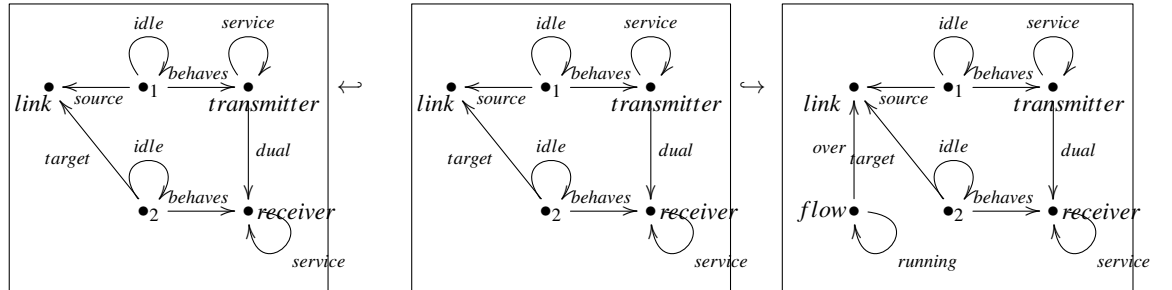


Fig. 13. Flow creation

In order to create a flow, I first need two things:

- I need to have a link between the nodes

- One node has to behave as the transmitter, while the other one should behave as the receiver.

Lets go through these rules one more time. The first rule is just telling us that, in order to be able to create a node, I need a connection between this node's (remember that this connection can be either logical or physical).

The second rule is telling us that, in order to create a flow of type X (lets say for example, that I want to create a ssh flow between this nodes), I need to guarantee that both nodes implements the same service, and one transmits while the other listens.

Finally, it's important to notice that the new flow is created in a running state, and is associated to the link with the link over, meaning that the flow is running over this link.

THINGS THAT HAD TO BE THOUGHT A BIT MORE:

1) Some rules may not be valid in this context ... How do I deny multiple ssh connections between same nodes? (Also, this sounds to me like a type condition, since another services may allow this behaviour, for example sql.)

- Remove flow from link

REMOVE_FLOW_FROM_LINK :

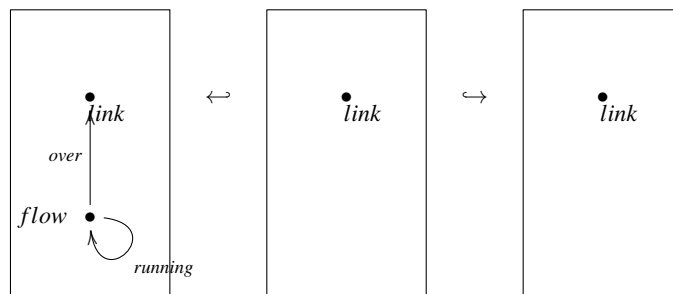


Fig. 14. Remove flow from link

This production is used to remove flows from links. Notice that it is not necessary to detach nodes from services, since they can still create new flows from the services where they are attached. This production could be used for example, when a flow ends.

- Flow's property creation

FLOW_PROPERTY_CREATION :

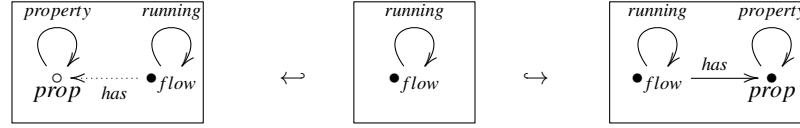


Fig. 15. Node properties creation

As other productions already defined, this one allow's us to create a new flow's property. The only condition to be satisfied, is that this new property to be created is not already created in flow.

QUESTIONS:

- 1) Can this properties be the actions associated in the nemo language ? Perhaps properties can be understood as premises that have to be guaranteed while the flow exist.
- 2) Some of this properties could be dynamic (this has sense, since flow is the representation of dynamism in the network).

– Flow's property deletion

REMOVE_FLOW_PROPERTY :

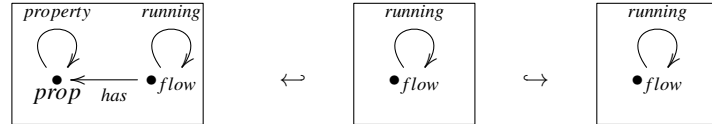


Fig. 16. Flow properties deletion

Finally, this productions allows us to remove properties from a flow.

4 Graph instantiation

In this section we will focus in giving several examples of the usage of the grammar recently defined. The examples used in this section will be those defined in a previous document (REMEMBER TO REWRITE EXAMPLES HERE)

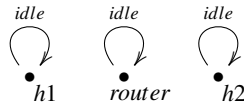
Lets first defined the first example, a network with two host and a router, which want to start a ssh connection.

There were two problems here:

- Neither of the host had ssh active
- Neither of the host were connected to the router.

Let's model our network, and see how the grammar defined before help us to detect problems in what we are trying to do.
This would be the network represented in our model:

FIRSTEXAMPLE



Having model our network, what we can check is that `NEW_FLOW` production can't be applied to my network (since there is no morphism between the left side of the production and this model). In this case, if I write a program for the situation described, we will have a compiler error, since there's going to be an instruction that cannot be done, and this is because the production associated with it cannot be executed.