A network computational model's for Software Defined Networking

Andrés Laurito[1] and Hernán Melgratti[2]

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

Abstract. In this paper we define a computational model for netowrk's programming languages 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. 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.
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 for networks. Finally we will show how this computational model is well defined, by using it to create a compiler for a new network progamming language recently emerged called Nemo.

1 Introduction

As it was said in [2] opendaylight link, sdn has multiple definitions as for example:

- Separation of control-plane and data-plane;
- Programmability;
- Defining the cloud networking;
- Network virtualization;
- Next-generation network management; and
- Network agility and operational simplicity.

In this paper, we intend to add a new defition: Network testing enviorment.

Nowadays networks have became 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

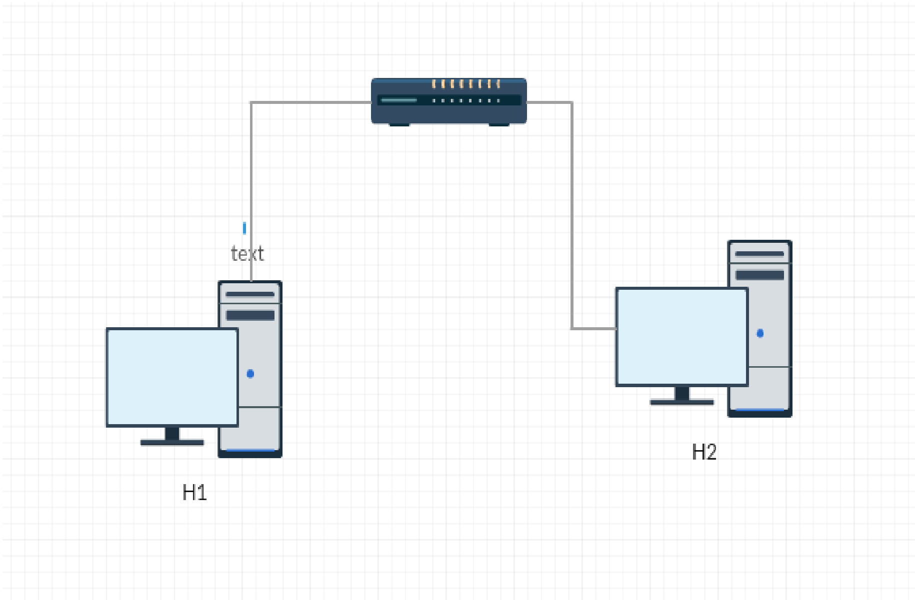
- **Debugging problem's:** It is rather difficult to debug problems in network's using most known tools such as ping, traceroute, netstat, between others.
- **Changeability problem's:** changing traffic rule's, or security rule's in network may impliy modifying hundred, or even thousands code line's distributed in several devices around the network.
- **Deployment problem's:** most changes are made mannually 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, betwen multiple others.
Missing this properties makes harder to build some tools for network programming languages, such as compiler's, debugger's, coding enviorment'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

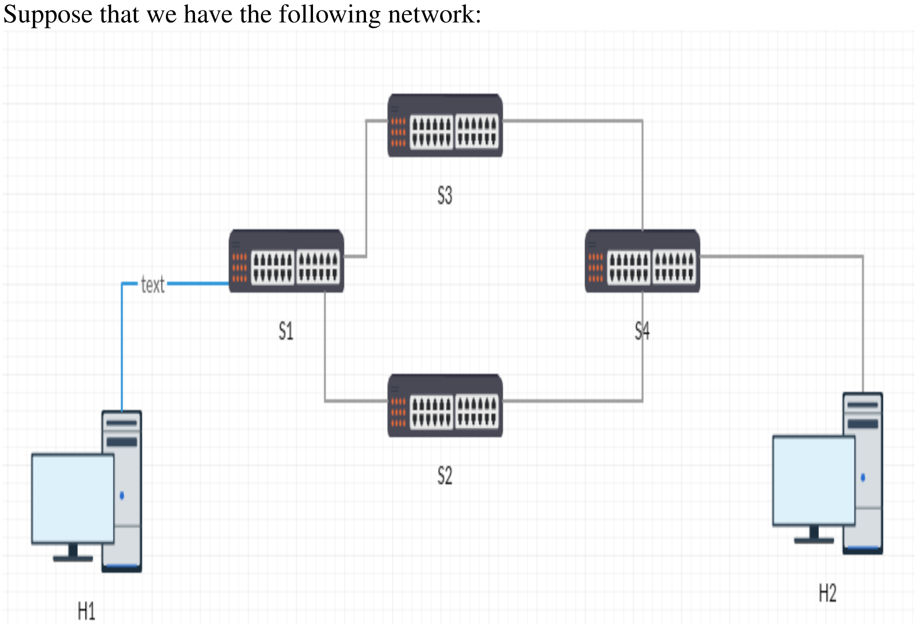
Let us remember that a program may be defined as a sequence of instruction's that executed perform a task in a computer or multiple computer's. Every program can be model as a Turing's machine or a finite lambda calculus computation, and despite of the model chosen, the operations are performed over 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:

<u>First Problem:</u>
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 fullfilled this connection.

Second Problem:
Suppose that we have the following network:



where H1 wants to establish a ssh connection with H2, using S1-¿S2-¿S4 path, and let's assume that there are not forwarding rules from S2 to S4.

The following examples have been extracted from [1]

Third Problem:

A tenant needs two connections to carry different service flows between two datacenters. One connection of the tenant is 40G bandwidth with less than 400ms delay, another connection is 100M bandwidth with less than 50ms delay.

Fourth Problem:

The tenant has two types of traffic, CDN sync traffic uses high bandwidth connection and online game traffic uses low latency connection.

Fifth Problem:

The tenant wants the online game traffic to go through WOC in nighttime before it is carried by low latency connection.

3 Resolving the problem

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 shall see in following sections). However, dynamic properties (such as applying some load balancing policy when a queue start's 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 a simulator, in order to check if the productions 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.

Solving the ATLAS problem with an intent would be as follows:

– First we will define the intent code as follows

```
Flow myControlFLow = new MultiCastFlow(src="10.0.0.1",dst=["10.0.0.2"
,"10.0.0.3","10.0.0.4"])

Action myLoadBalancing = new Action({Lo que recibiria este como
 constructor, lo pienso como un lambda, seria un bloque de codigo que va a
 estar ejecutandose en el contexto del controller})

Intent "multicast ATLAS"
        Type "HostToMultiHostIntent"
        Select myControlFlow
        Action myLoadBalancing
```

– This intent will be taken by the compiler, who will parse the intent, evaluate if it's both syntactically and semantically correct and finally produce a request to the ONOS api as follows (it is possible that the request will have more attributes defined in it, but this will depend on the intent, for example, in the action defined, if a condition is set into the intent, etc).

```
curl --X POST {
        "type":"HostToMultiHost",
        "appid":"org.onosproject.ovsdb",
        "priority":"55",
        "source":"10.0.0.1",
        "destinations":["10.0.0.2","10.0.0.3","10.0.0.4"]
}
```

– This request will be recieved by the intent's api which is defined in the ONOS controller. The api will first decode the json, build the intent class associated to the intent recieved and finally will compile this class. This final "compilation" means to create the necessary objects to make this intent work (for example, to create pathsIntents for each (source,destination) pair). The classes implemented in the ONOS intent's api in order to make this post work are the following ones:
  - HostToMultiHostIntent.java . This class is the one who has all the information needed to build the intent.
  - HostToMultiHostIntentCodec.java . This class extends JsonCodec, and is in charge of decoding the json received by the POST. Once the JSON has been decoded, this class builds an instance of HostToMultiHostIntent
  - HostToMultiHostIntentCompiler.java . This class is in charge of the logic of building the intent into the controller. It's principal functionallity is to create every object necessary to make the intent work. In this multicast examples, some of this objects are the paths from the source to each of the destinations defined, to translate the actions into real objects who will make the job done, etc.
  - HostToMultiHostIntentCompilerTest.java

  - HostToMultiHostIntentTest.java
- Once the intent has been compiled, what happens now is that ONOS starts communicating with the Floodlight controller (which is included in ONOS as compiled libraries), who will be the one responsible of loading the necessary OpenFlow entries in each of the SDN switches in the network topology.
- Since the main objective here is to not deploy until we are really sure that everything works fine, floodlight will communicate with a simulator, instead of a real network. With this approach, we can manage to test the entire intent in a controlled network. To make this happens we have: Cual de las dos opciones ser la mejor?
  - created a DEVS specification of a SDN swtich, which implements the minor set of OpenFlow messages in order to make the intent work.
  - created a DEVS specification of a SDN switch, where the OpenFlow protocol is implemented by a third process which is not inside the simulator. This process is the one who interacts with Floodlight, and is constantly comunnicating with the simulation (like Mininet does with OpenVSwitch).

4 Semantic's rules

In this section we introduce the semantic's rules used in Haikunet.

- Array semantic's rules

$$\frac{}{\Gamma \triangleright [\,] : \text{Array}[\Phi]} \text{ Array definition}$$

(Acá Φ es o Device o Link o Flow)

$$\frac{\Gamma \triangleright \text{x:Elem}, \ \Gamma \triangleright \text{y:Array}[\Phi]}{\Gamma \triangleright \text{ y + x : Array}[\Phi]} \text{ Adding elements to Array}$$

- Definition semantic rule

$$\frac{\Gamma \triangleright \text{ y} : \Phi}{\Gamma \triangleright \text{ x := y} : \Phi}$$

- Link semantic's rules

$$\frac{\Gamma \triangleright \text{x:Bandwith}, \ \Gamma \triangleright \text{y:ID}, \ \Delta \triangleright \ tag_1, tag_2\text{:TAG}}{\Gamma, \Delta \triangleright \text{Link}(tag_1 = \text{x}, tag_2 = \text{y}) : \text{Link}} \text{ Link definition}$$

- Device semantic's rules (A device can be either a Host or a Router)

$$\frac{\Gamma \triangleright \text{x:Device\_ID},\ \Delta \triangleright\ tag_1\text{:TAG}}{\Gamma, \Delta \triangleright \text{Host}(tag_1 = \text{x}) : \text{Host}}\ \text{Host definition}$$

- Action semantic's rules

?????

- Condition semantic's rules

?????

- Flow semantic's rules

$$\frac{\Gamma \triangleright \text{w, x : Device},\ \Gamma \triangleright \text{y:ID},\ \Gamma \triangleright \text{z:Priority},\ \Delta \triangleright\ tag_1, tag_2, tag_3, tag_4\text{:TAG}}{\Gamma, \Delta \triangleright \text{Flow}(tag_1 = \text{w},\ tag_2 = \text{x},\ tag_3 = \text{y},\ tag_4 = \text{z}) : \text{Flow}}$$

$$\frac{\Gamma \triangleright \text{w: Array[Device]},\ \Gamma \triangleright \text{x:Device},\ \Gamma \triangleright \text{y:ID},\ \Gamma \triangleright \text{z:Priority}\ \Delta \triangleright\ tag_1,\ tag_2,\ tag_3,\ tag_4\text{:TAG}}{\Gamma, \Delta \triangleright \text{Flow}(tag_1 = \text{w},\ tag_2 = \text{x},\ tag_3 = \text{y},\ tag_4 = \text{z}) : \text{Flow}}$$

$$\frac{\Gamma \triangleright \text{w: Device},\ \Gamma \triangleright \text{x:Array[Device]},\ \Gamma \triangleright \text{y:ID},\ \Gamma \triangleright \text{z:Priority}\ \Delta \triangleright\ tag_1,\ tag_2,\ tag_3,\ tag_4\text{:TAG}}{\Gamma, \Delta \triangleright \text{Flow}(tag_1 = \text{w},\ tag_2 = \text{x},\ tag_3 = \text{y},\ tag_4 = \text{z}) : \text{Flow}}$$

- Intents semantic's rules

$$\frac{\Delta \triangleright \text{x:TAG},\ \Gamma \triangleright \text{y:Flow}}{\Gamma, \Delta \triangleright \text{Intent x Select y : Intent}}$$

$$\frac{\Delta \triangleright \text{x:TAG},\ \Gamma \triangleright \text{y:Flow},\ \Gamma \triangleright \text{z:Action}}{\Gamma, \Delta \triangleright \text{Intent x Select y Action z : Intent}}$$

$$\frac{\Delta \triangleright \text{w:TAG},\ \Gamma \triangleright \text{x:Flow},\ \Gamma \triangleright \text{y:Action},\ \Gamma \triangleright \text{z:Condition}}{\Gamma, \Delta \triangleright \text{Intent w Select x Action y Condition z : Intent}}$$

5 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 topoloy:

- Creating a node

NEW_NODE :

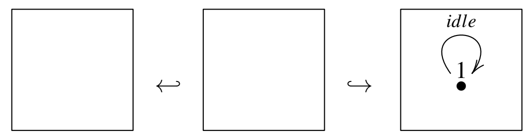


Fig. 1. Creating a node

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 assocciated address) or logic (this mean's a group of physical nodes, meanning 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 :

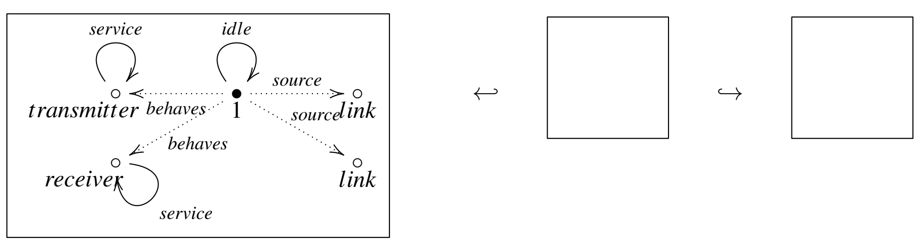


Fig. 2. Deleting a node

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 :

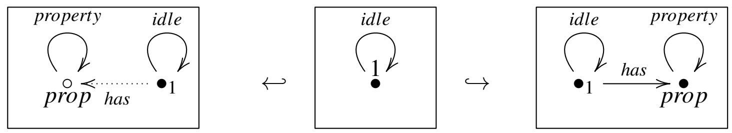


Fig. 3. Node properties creation

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 :

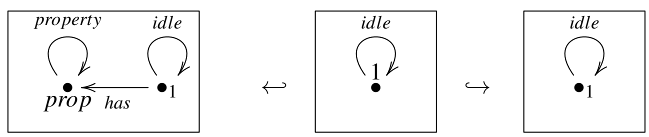


Fig. 4. Node properties deletion

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

NEW_SERVICE :

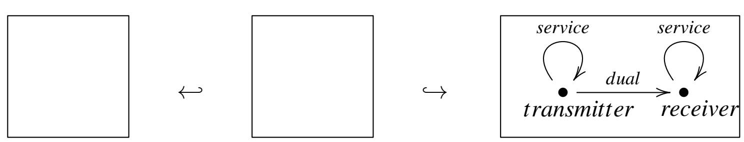


Fig. 5. Service Creation

In this production we define a new service. A service is either some property implemented in nodes, or implemented in another node, wich 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 reicever in the graph.

– Service Deletion

REMOVE_SERVICE :

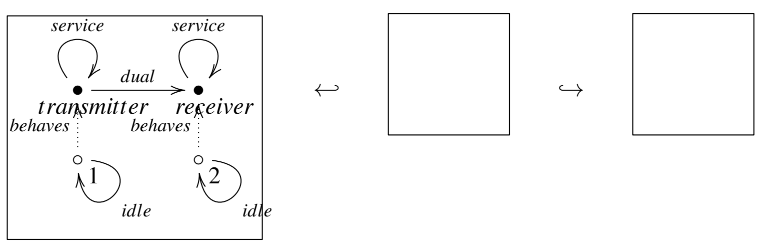


Fig. 6. Service Creation

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 endpiont of communication

NODE_IMPLEMENTS_ROLE :

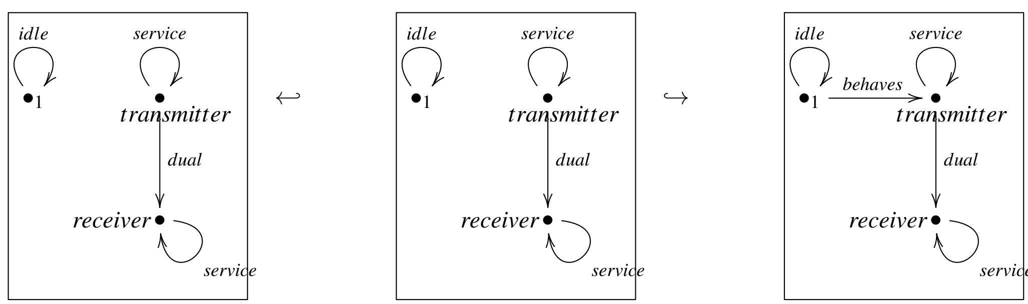


Fig. 7. Node implements role

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 endpiont in the communication

NODE_STOPS_IMPLEMENTING_ROLE :

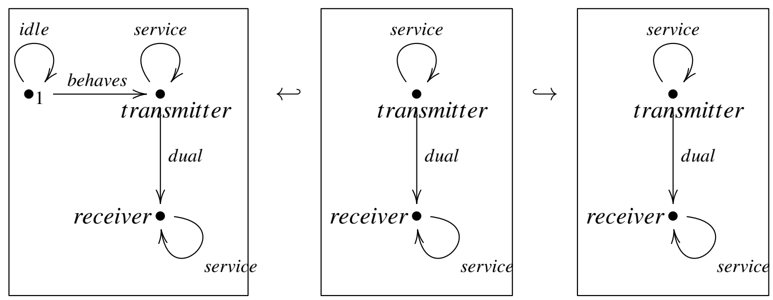


Fig. 8. Node stops implementing role

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

NEW_LINK :

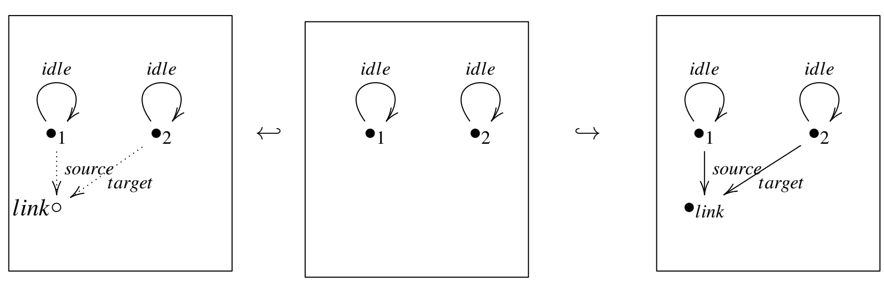


Fig. 9. Link creation

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 likes a wildcard, meaning that from this state, multiply actions can be applied to the node.

- Link deletion

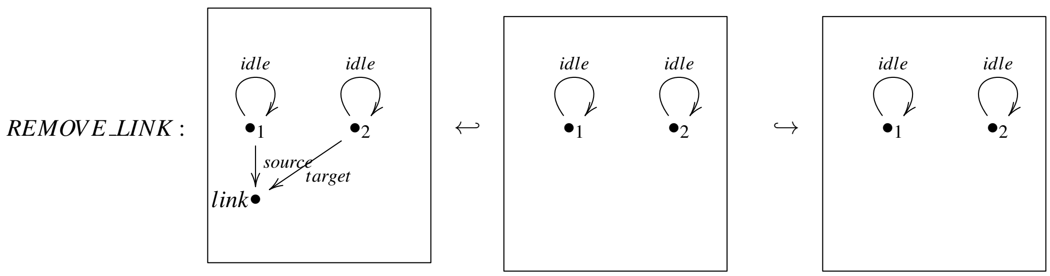


Fig. 10. Link deletion

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

- Link's property creation

NEW_LINK_PROPERTY :

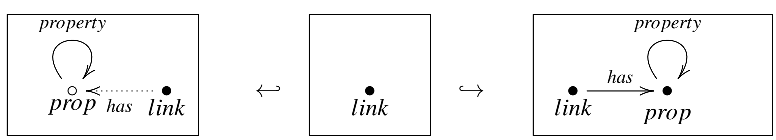


Fig. 11. Link's property creation

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 bandwith, average latency, perhpahs 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?. Perphas it happens something similar as nodes, that we need some 'idle' state, for example, an empty state).

– Link's property deletion

REMOVE_LINK_PROPERTY :

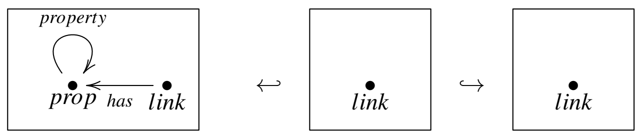


Fig. 12. Link's property deletion

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 :

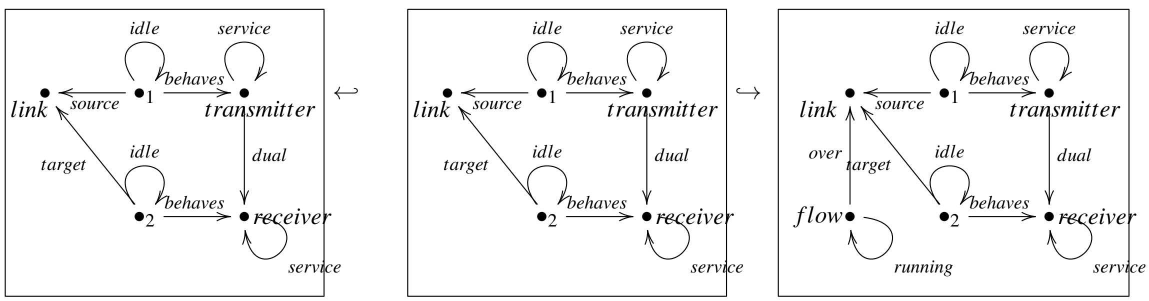


Fig. 13. Flow creation

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 reicever.

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 conection 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.

THINS THAT HAD TO BE THINK A BIT MORE:
1) Some rules may not be valid in this context ... How do I deny multiple ssh conections betwen 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

$REMOVE\_FLOW\_FROM\_LINK$:

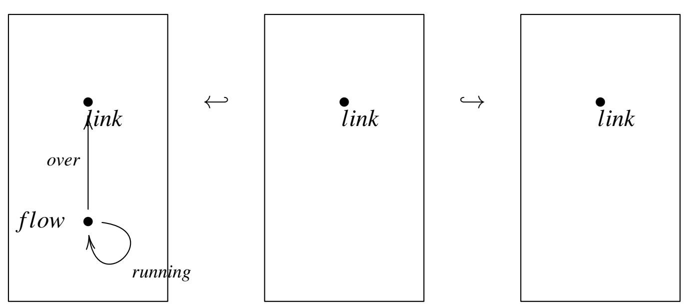


Fig. 14. Remove flow from link

This production is used to remove flows from links. Notice that 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 :

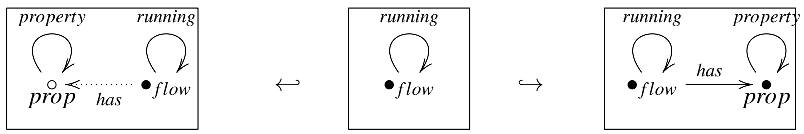


Fig. 15. Node properties creation

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 ? Perphaps 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 :

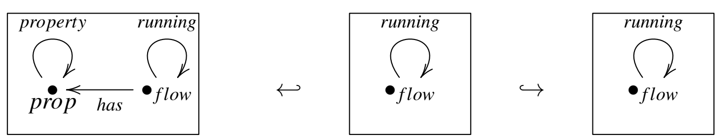


Fig. 16. Flow properties deletion

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

6 Graph instantiation

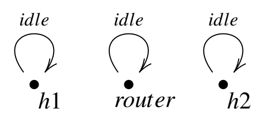
In this section we will focus in giving several examples of the usage of the grammar recently defined. The examples used in this section will be those defined in a previous document (REMEMBER TO REWRITE EXAMPLES HERE)
Lets first defined the first example, a network with two host and a router, which want 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.

References

1. Y. Xia, Ed., S. Jiang, Ed. T. Zhou, Ed. and S. Hares NEMO (NEtwork MOdeling) Language draft-xia-sdnrg-nemo-language-00 Internet-Draft
2. Madhu Venugopal OpenDayLight https://www.opendaylight.org/news/blogs/2013/12/opendaylight-developer-spotlight-madhu-venugopal