3 asteriscos \*\*\* indicam que uma seção está incompleta.

**Description of Command and Control Networks in Coq**

Guilherme Gomes Felix da Silva

guilhermegfsilva@gmail.com

Department of Informatics, PUC-Rio

Advisor: Dr. Edward Hermann Haeusler

July 2021

**Abstract**

**1. Motivation**

A command and control network is any system in which individuals or entities which possess authority over other individual may apply that authority with the aim of achieving a certain objection. While the term can be used in various contexts, it is commonly used in reference to a military system. **[source?]** \*\*\*

Apresentamos aqui uma definição de uma rede de controle em Coq. \*\*\*

Para demonstrar como aplicar os axiomas que definimos, vamos definir primeiro definir exemplos de redes. Estas redes consistem de três tipos de objetos: nós, locais (cada local sendo associado a um único nó), e áreas (que podem conter vários nós ou locais).

**2. Representation of a Network in Coq**

**2.1. Relevant Concepts**

**Nodes**

The nodes represent the individuals in a C2 network. The hierarchy between these individuals is represented by a connected and acyclic graph, i.e., a tree. The root of the tree represents the leader in our C2 system, with each edge indicating which individuals are direct subordinates of which. By definition, we have established that each individual has only one direct superior. Figure 1.2 shows an example of a graph representing such a system, with individual 1 as the leader, 2 and 3 as its direct subordinates, and so on.

**Fig. 1.2:** Directed graph representing the hierarchy in a command and control network.

**Defining a Network**

Now, let us describe how to represent these C2 graphs in Coq. To do so, we need to define a data structure. Coq provides us with the means to do so by the Structure operator.

Require Import Coq.Lists.List.

Section nets.

Structure net : Type := {

nodes : nat ;

leader : nat ;

superior : list (nat \* nat) ;

second\_in\_command : nat := get\_second superior leader ;

parent : nat -> nat := get\_parent superior ;

children : nat -> list nat := get\_children superior ;

is\_parent : nat -> nat -> Prop := is\_parent\_func superior;

is\_parent\_bool : nat -> nat -> bool := is\_parent\_func\_bool superior;

node\_order : nat -> nat := get\_node\_order ;

sorted\_superior : list (nat \* nat) := sort superior ;

node\_level : nat -> nat := get\_level superior ;

}.

Here, we have defined a command and control network as a structure with a single leader node and a set of nodes subordinate to this leader, who in turn can have their own subordinates, and so on. The objects and functions that make up this structure are:

* The number of nodes in the network, defined here as nodes, which are represented by a single natural number. By definition, nodes in our model are numbered individually starting from 1 without skipping any number, so nodes will also always be equal to the highest node value in a particular network. A structure with a value of 10 assigned to the nodes field, for example, will have a total of nodes numbered from 1 to 10.
* leader tells us the index of the node which is the network’s leader, equivalent to the root of the graph.
* superior tells us which nodes are direct subordinates of which others. This field is a list of pairs of natural numbers representing our graph, which each pair being a single edge containing the index of two nodes (parent and child). We assume that the numbers contained within these pairs are consistent with the node values defined by nodes.
* second-in-command is a function which tells us which node in the network is the second-in-command of the current leader and the one that should replace the current leader if necessary. It is defined as the first subordinate of the leader node, as we will describe in more detail ahead.
* parent and children are functions that receive a single node (natural number) as an argument and, respectively, return the index of the superior/parent node or a list of indices indicating the children/subordinates of the node.

Note that all we have defined so far are the headers of the functions in our structure, which tell us what types they receive as arguments and and which types they should return. For example, look at the definition of parent here:

parent : nat -> nat := get\_parent superior ;

We are informing Coq after the : operator that the function receives a single natural number value and also returns a natural number value. After the := operator, we tell Coq how the computation of the return value is to be done—in this case, by calling a function named get\_parent which we will define later outside of our data structure. Since this function will require the values stored in superior, the list of edges, we give that as an argument here as well.

Now, let us get into the actual implementation of these individual functions.

The second-in-command function, as stated, tells us which node is considered the highest-ranking subordinate of the current leader and the one that should be made the leader if the current one needs to be replaced. \*\*\*

Fixpoint get\_second (edges : list (nat \* nat)) (leader : nat) : nat :=

match edges with

| (a,b) :: edges' => if (Nat.eqb a leader)

then b

else get\_second edges' leader

| nil => 0

end.

The get\_parent receives one node and needs to tell us its parent node. To do this, it recursively searches through the list of edges, comparing the second number in each one (the child node, or b) with the given value until it finds a match. When that happens, the first value of the pair (the parent node, or a) is returned. Since our model already assumes that the network is defined with each node having only one parent, there is no need to search through the rest of the list after a match is found.

Should the function finish searching the list without finding an edge whose target node matches the given value, it returns 0 by default, indicating that the node has no parent. This should happen only when the value given is the leader, i.e., the root node.

Fixpoint get\_parent (edges : list (nat \* nat)) (node : nat) : nat :=

match edges with

| (a,b) :: edges' => if (Nat.eqb node b) then a

else get\_parent edges' node

| nil => 0

end.

The get\_children function operates similarly to get\_parent. However, since a node can have any number of children, this function needs to return a list of natural numbers. Once again, the function recursively calls itself to Search through the list of edges, this time comparing the given value with the parent node, a, in each edge. Once a match is found, we append the child value b to the list that will be our final product and continue searching via recursion, as you can see below. At the end of the run, we will have searched through every edge and have the complete list of the node’s children.

If the node has no children, an empty list value nil will be returned.

Fixpoint get\_children (edges : list (nat \* nat)) (node : nat) : list nat :=

match edges with

| (a,b) :: edges' => if (Nat.eqb node a) then b :: get\_children edges' node

else get\_children edges' node

| nil => nil

end.

We have also defined functions that tell us *if* two given nodes are parent and child to each other. This question of “if” is answered in Coq by two different types, proposition (Prop) or boolean (bool).

Note the importance of capitalization here. In Coq, False and True with capital letters are interpreted as values of type Prop, while false and true are interpreted as values of type bool.

Fixpoint is\_parent\_func (edges : list (nat \* nat)) (a b : nat) : Prop :=

match edges with

| nil => False

| h :: t => h = (a,b) \/ is\_parent\_func t a b

end.

Fixpoint is\_parent\_func\_bool (edges : list (nat \* nat)) (a b : nat) : bool :=

match edges with

| nil => false

| h :: t => ((Nat.eqb (fst h) a) && (Nat.eqb (snd h) b)) || is\_parent\_func\_bool t a b

end.

Next, let us look at the get\_level function. This function tells us the level of a node in the hierarchy. The leader’s level is by definition 1, while the level of its direct subordinates is 2, and so on.

Fixpoint get\_level\_run\_once (edges : list (nat \* nat)) (node\_and\_depth : nat \* nat) : nat \* nat :=

match edges with

| (a,b) :: edges' => if (Nat.eqb b (fst node\_and\_depth)) then get\_level\_run\_once edges' (a, (snd node\_and\_depth) + 1)

else get\_level\_run\_once edges' ((fst node\_and\_depth), (snd node\_and\_depth))

| nil => ((fst node\_and\_depth), (snd node\_and\_depth))

end.

Definition get\_level\_run\_once\_result (edges : list (nat \* nat)) (node : nat) : nat \* nat :=

get\_level\_run\_once edges (node, 1).

Fixpoint get\_level\_run\_all (edges : list (nat \* nat)) (times : nat) (node\_and\_depth : nat \* nat) : nat \* nat :=

match times with

| 0 => ((fst node\_and\_depth), (snd node\_and\_depth))

| S n => get\_level\_run\_all edges n (fst (get\_level\_run\_once edges node\_and\_depth),

snd (get\_level\_run\_once edges node\_and\_depth))

end.

Definition get\_level (edges : list (nat \* nat)) (node : nat) : nat :=

snd (get\_level\_run\_all edges (length edges) (node, 1)).

\*\*\*Restante das implementações de funções\*\*\*

**Defining a Network Instance**

**Fig. \_.\_:** Network instance example.

Definition net\_1 : net := Build\_net 7 1

((1 , 2) :: (1 , 3) :: (2 , 4) :: (2 , 5) :: (3 , 6) :: (3 , 7) :: nil)

((1 , 2) :: (1 , 3) :: (2 , 4) :: (2 , 5) :: (3 , 6) :: (3 , 7) :: nil).

**Defining Properties**

Agora, podemos definir propriedades sobre uma rede e seus elementos.

Começaremos definindo a propriedade de que o nó raiz de qualquer rede sempre será um de seus elementos. A lista de elementos, ou nós, é representada por um número natural que indica a quantidade de nós em uma rede. Portanto, um valor 10 atribuído ao parâmetro nodes indica que uma rede possui 10 nós numerados de 1 a 10. Para afirmarmos que a raiz é sempre um dos nós, basta dizer ao provador que o índice dela está contido neste intervalo.

Isto é, queremos afirmar que

Esta definição é feita em Coq de forma bem simples:

Definition leader\_is\_in\_net := forall n : net, leader n <= nodes n.

Outra propriedade que devemos definir afirma que nenhum nó pode ser seu próprio superior. Para isto, utilizamos o elemento superior, que é uma lista de pares de números naturais correspondes às arestas de um grafo que indica quais nós da rede possuem uma relação de superioridade direta com quais outros nós.

Definition no\_self\_superior :=

forall (n : net) (i : nat), fst (nth i (superior n) (0,0)) <>

snd (nth i (superior n) (0,0)).

* superior n é a função superior aplicada a uma rede n qualquer, que retorna a lista de arestas.
* nth i (superior n) (0,0) retorna o i-ésimo elemento desta lista, ou seja, uma única aresta. O terceiro argumento desta função, definido aqui como o par (0,0), é um valor de retorno “default” que indicamos que a função nth deve retornar caso o valor de i informado seja inválido para a lista em questão (ou se a lista estiver vazia?)
* fst e snd são funções que retornam o primeiro e o segundo elementos de um par, respectivamente, e o operador <> indica que dois valores são necessariamente diferentes. Portanto, o que estamos definindo aqui é simplesmente que a lista de arestas superior n não pode possuir um elemento (a,b) em que a e b sejam iguais.

Aplicando estes mesmos princípios, podemos definir outras especificações sobre os elementos de uma rede.

Definition leader\_is\_top := forall (n : net),

~ exists i : nat, snd (nth i (superior n) (0,0)) = leader n.

(\*Definition is\_superior\_to (n : net) (a b : nat)\*)

Podemos também definir funções que possam ser aplicadas a uma rede e seus elementos. Veja como definimos uma função que informa o número de subordinados de um nó:

Fixpoint num\_children (edges : list (nat \* nat)) (node count : nat) : nat :=

match edges with

| nil => count

| (a,b) :: edges' => if (Nat.eqb a node)

then num\_children edges' node (count+1)

else num\_children edges' node count

end.

* Os \*\*\*

Caso seja necessário, podemos também definir funções que realizem alterações na rede. Um exemplo de uma operação que podemos representar diz respeito a como a rede deve reagir caso haja necessidade de substituir o indivíduo que a comanda.

Fixpoint change\_leader (edges : list (nat \* nat)) (old new : nat) : list (nat \* nat) :=

match edges with

| nil => nil

| (a,b) :: edges' => if (Nat.eqb a old) && (Nat.eqb b new)

then change\_leader edges' old new

else if (Nat.eqb a old)

then (new,b) :: change\_leader edges' old new

else if (Nat.eqb b old)

then (a,new) :: change\_leader edges' old new

else (a,b) :: change\_leader edges' old new

end.

End nets.

* Os \*\*\*

**Conclusion**

**References**

1. Chlipala, Adam. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press. 2013.

2. Alberts, David S. Hayes, Richard E. *Understanding Command and Control*. CCRP Publication Series. 2006.