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).

área U

P

R

Q

S

T

**Fig. 1.1:** Exemplo de rede com hierarquia entre nós.

**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.

Require Import Coq.Lists.List.

Section nets.

Structure net : Type := {

nodes : nat ;

leader : nat ;

superior : list (nat \* nat) ;

knows : 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 nodes (length superior);\*)

node\_level' : nat -> nat := get\_level\_aux superior 1 ;

}.

Definimos aqui uma rede de comando e controle como uma estrutura que possui um nó líder e um conjunto de nós subordinados a este líder, que por sua vez podem possuir seus próprios subordinados, e assim por diante. São os objetos e funções que compõem esta estrutura:

* Os nós da rede, denominados aqui como nodes, são representados por um número natural. Por exemplo, o valor 10 indica que temos uma rede com 10 indivíduos.
* leader indica o índice do nó que é o líder da rede, equivalente à raiz do grafo que a representa.
* superior indica a relação de superior e subordinado entre os nós. O grafo correspondente a cada relação é definido como uma lista de pares de números naturais, com cada par correspondendo a uma aresta direcionada e contendo o índice de dois nós (origem e destino).
* second-in-command é uma função que informa qual nó da rede é o segundo em comando, e portanto, aquele que deve assumir o comando caso o líder seja eliminado.
* parent e children são funções que recebem um nó (número natural) como parâmetro e retornam, respectivamente, o índice dó nó pai (superior) ou uma lista de índices correspondentes aos filhos (subordinados) do nó;

Note que, até agora, definimos apenas os cabeçalhos das funções pertencentes à estrutura, indicando os tipos que elas recebem como parâmetro e os tipos que retornam. Vejamos como se faz a implementação propriamente dita destas funções.

A função second-in-command \*\*\*

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.

A função get\_parent percorre a lista de arestas em busca de \*\*\*. Como a nossa especificação de rede já tem como parte de sua definição que cada nó possui apenas um nó pai, \*\*\*

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.

A função get\_children opera de forma semelhante a get\_parent. Porém, esta função deve retornar uma lista de números naturais. Novamente, a função chama a si mesma recursivamente para percorrer a lista arestas, comparando o nó dado com a origem de cada aresta (a). Caso a seja equivalente ao nó, utilizamos o operador de concatenação :: para anexar o nó filho correspondente, b, à lista de números que queremos formar. Ao final da chamada recursiva, teremos varrido todas as arestas e definido a lista completa de subordinados do nó.

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.

Definimos também funções que recebem dois nós como parâmetros e informam se esses dois nós são pai e filho na rede. \*\*\*

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.

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

**Defining a Network Instance**

**Fig. \_.\_:** Exemplo de instância de rede.

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 \*\*\*

**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.