Description of Command and Control Networks in Coq

**Guilherme G. F. da Silva1, Edward Hermann Haeusler2 and Cláudia Nalon3**

1 Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

2 Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil

3 University of Brasília, Brasília, DF, Brazil

guilhermegfsilva@gmail.com, hermann@inf.puc-rio.br, nalon@unb.br

***Abstract.*** A command and control (C2) system can be defined as any group of individuals organized hierarchically in which higher-ranking individuals can issue directions to their subordinates with a certain goal in mind. We present a model for representation of command and control networks in the Coq proof assistant based on a tree data structure. Our model utilizes Coq’s implementation of data structures and includes examples of how to define functions and properties that may be relevant in a C2 system.

1. Introduction

The term “command and control” can be used in various contexts, with one common example of its use being a military system. More generally speaking, a command and control network can be defined as any system in which an individual or entity may issue directions to another with the aim of achieving a certain objective. There are countless ways in which a C2 system may be organized, but the work described here concerns itself specifically with systems organized as a hierarchy, with individuals subordinate to others which may give them directions.

Naturally, there is a variety of algorithms, properties and functions that may be relevant when discussing a C2 system. For example, knowing which individual ranks the highest in the network (i.e. the leader), we may be interested in knowing who its direct subordinates are, and out of these, which one should take control in the event that the leader is eliminated. Alternately, we may want to guarantee that the way the network is organized makes sense, such as by not having two individuals be subordinate to each other—that is, that there are no cycles. We may also want to make sure that there are no “orphaned” individuals in the network, i.e., that every node other than the leader is subordinate to some other node.

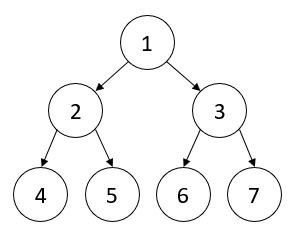
Our work described in this paper is a model for generic representation of a C2 network in the Coq proof assistant. We have done this by first establishing how to represent the network itself—specifically, as a tree data structure—followed by the definition of relevant variables and functions, and lastly, how these functions are implemented. Our goal is to both help Coq developers seeking some insight into how certain types of algorithms are done in the proof assistant, and to provide some useful tools for Coq projects dealing with anything related to C2 or hierarchy.

The Coq source code developed for this project can be found at http://github.com/GGFSilva/CommandAndControl. Some parts of this summary will reference the source code for a more in-depth explanation.

1. Representation of a Network in Coq
   1. Basic Concepts

Firstly, let us establish what networks mean in this context. We have defined a network as a group of nodes, with each node representing an individual in a C2 system. 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 in the network (other than the leader) has only one direct superior. However, an individual can have any number of direct subordinates. Additionally, every network has one and only one leader. Fig. 1 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. Directed graph representing the hierarchy in a command and control network.**

* 1. 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 via the **Structure** operator. The Coq code containing the main body of the structure is defined as **net** in our source code. This **net** object represents a command and control network as a structure type containing 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 as follows.

The number of nodes in the network is defined as **nodes**, a single natural number. By definition, nodes in our model are numbered individually starting from 1 without skipping any number, so the value of **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 10 nodes numbered from 1 to 10.

**leader** is a natural number value which 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, with each pair representing a single edge of the graph via the indices of two nodes (parent and child). We assume that the numbers contained within these pairs are consistent with the node values defined by **nodes**. For example, for the network shown in Fig. 1, our list of edges would be represented in Coq as the list **(1,2) :: (1,3) :: (2,4) :: (2,5) :: (3,6) :: (3,7) :: nil**.

These three are the fields that must be given as parameters when creating an instance of the structure, as we will see later. The remaining fields are the functions our structure will use.

**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.

**is\_parent** and **is\_parent\_bool** are two similar functions that tell us if two given nodes are parent and child to each other in the graph.

Lastly, **node\_level** tells us the level of a node in the hierarchy. By definition, the leader should have a level value of 1, its direct subordinates should have a level of 2, and so on.

The next section gives some explanation of how the actual implementation of these functions is done.

* 1. Network Functions

Second\_in\_command. This 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. We define the second-in-command node as the first node to appear in the list of edges as a direct subordinate of the leader node. The function that returns this node is defined in our code as **get\_second**. Note how the function receives two parameters, **edges** and **leader**. This is consistent with how we defined **second\_in\_command** in the structure, i.e. that it is always **get\_second** applied to two arguments, the list of edges and the leader value.

What we are doing here is recursively searching through the list of edges, comparing the first number in each one (the parent node, or **a**) with the value of **leader** until it finds a match. When that happens, the second value of the pair (the child node, or **b**) is returned. If the value does not match, we call **get\_second** again recursively on the remaining edges, defined here as **edges'**. If we reach the end of the list without finding any matches, we return a default value of 0 indicating that no valid second-in-command node was found.

Parent. This function receives one node and needs to tell us its parent node. Once again, we find the value by searching through the list of edges recursively, this time comparing the node value with the child node in each edge. The **get\_parent** function in our code does this. 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.

Children. This function operates similarly to **parent**. However, since a node can have any number of children, this function needs to return a list of natural numbers. The **get\_children** function returns this list. Once again, the function works by recursively calling itself to search through the list of edges, this time comparing the given value with the parent node, **a**, in each edge. If 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. If there is no match, we do a recursion without appending anything to the list. At the end of the run, we will have searched through every edge and have the complete list of children of the given node. If the node has no children, an empty list value **nil** will be returned.

Is\_parent and is\_parent\_bool. These functions tell us if two given nodes are parent and child. This question of “if” can be represented in Coq by two different types, proposition (Prop) or boolean (bool). For comparison’s sake, we have included two different “is parent” functions, one for each of these types. As you will see, they are mostly similar but with some differences in which operators are used.

Note the importance of capitalization in the names of certain constants 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**. In our source code, **is\_parent\_func** and **is\_parent\_func\_bool**, respectively, are the functions that contain the code for implementing the two preceding functions proper. As you can see in the code, both functions work by searching through the edge list for an element in which both of the values, **a** and **b**, match the given parent and child. A value of “false” is only returned if the function searches through the entire list without finding any matches.

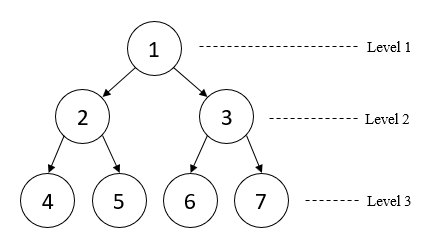
Node\_level. This function is a bit more complex. It needs to tell us the level of a node in the hierarchy. The leader’s level is by definition 1, while the level of its direct subordinates is 2, and so on.

To tell the level of a node, we need to count how many levels separate it from the leader. We do this by first searching the edge list for the pair **(a,b)** where **b** is the value of our target node. Once we have found it, we increment a counter by 1 and call a recursion to find the level of **a**, its parent node, and we continue doing this until **a** matches the value of the leader, who we already know is the root. Our counter will then let us know how many levels separate the target node from the leader. In summary, we search backwards starting from our target node and count the number of levels toward the root.

The issue here is that since we do not know how the edges are ordered, we need to run through the list multiple times. In other words, this is a function with O(n²) complexity in which we need to search the list at least **n** times to guarantee we will have the value we want. Doing this requires more than one level of recursion as shown in the **get\_level** function in our code

* 1. Defining a Network Instance

Now that we have talked at length about our structure and its functions, we can create an actual instance of a network to see some of them in use. Revisiting the network example shown earlier in Fig. 1, we can see that it has three different levels of hierarchy.



**Fig. 2. Example of a network with seven individuals and three hierarchy levels.**

By creating this network in Coq as the object **net\_1**, we can try computing the functions we had previously defined on it and confirm that the results we get are the expected ones. Table 1 shows some examples of this, with the left column showing Coq’s Compute command being applied to **net\_1** with the corresponding functions and nodes as parameters, and the right column showing the corresponding output displayed by Coq.

**Table 1. Examples of Coq operations on a network instance.**

|  |  |
| --- | --- |
| Coq Input | Coq Output |
| Compute is\_parent\_bool net\_1 1 2.  Compute is\_parent\_bool net\_1 2 3.  Compute node\_level net\_1 1.  Compute node\_level net\_1 2.  Compute node\_level net\_1 3.  Compute node\_level net\_1 4.  Compute parent net\_1 2.  Compute children net\_1 1. | = true : bool  = false : bool  = 1 : nat  = 2 : nat  = 2 : nat  = 3 : nat  = 1 : nat  = [2; 3] : list nat |

* 1. Defining Properties

In this next part, we will talk about how to use the appropriate tools in Coq to define properties that a network and its elements must have. Properties like this are the sort of thing that would be used as a basis when building and proving theorems related to command and control systems.

To start with a simple example, let us define the property that “in any network, the leader must be one of its elements”. As mentioned before, the list of nodes in a network is represented by a single natural number telling us how many nodes there are, with the assumption that they are all individually numbered from 1 to the stated value. Therefore, a value of 10 in this field, for example, tells us that we have a network with 10 nodes numbered 1 to 10. The leader is also represented by a natural number. Thus, in order to define that the leader is always a valid node, all we need to do is inform Coq that its index is contained in that interval. This can be done in Coq via the simple definition line **Definition leader\_is\_in\_net := forall n : net, leader n <= nodes n.**

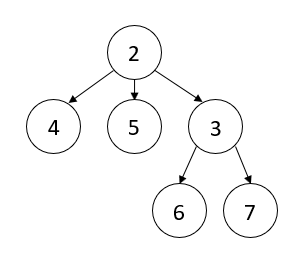
Another property we can define is the affirmation that no node can be its own superior. To do this, we will use the **superior** element, which, as already shown, is a list of pairs of numbers representing each edge of the graph, i.e., the indices of a parent node and child node. Basically, what we want to say is that none of these pairs contain the same number twice. This can be done via the definition line **Definition no\_self\_superior := forall (n : net) (i : nat), fst (nth i (superior n) (0,0)) <> snd (nth i (superior n) (0,0)).**

This way, one can easily define any property pertaining to all networks and use it to build theorems and proofs.

1. Dynamic Networks

One thing we have not yet explored in our Coq model is the fact that a C2 network may need to undergo changes in its organization over time. For example, an individual who has been removed from the network may need to be replaced or have its subordinates transferred to another.

We can implement these changes into our model by writing functions that can be applied to a network to alter the configuration of its elements. Consider the network shown previously in Figs. 1 and 2, for example. This network has node 1 as its leader and 2 and 3 as the leader’s direct subordinates. According to the function described in section 2.3, node 2 is the one considered this network’s second-in-command. So let us consider what might happen if node 1 were eliminated and needed to be replaced with its second-in-command, i.e. 2. Naturally, we need to account for node 1’s other direct subordinates (in this case, node 3). One way to handle this is to have the subordination of the other nodes transferred directly to 2, the node that succeeds 1. Fig. 3 shows the resulting network that we expect from this transformation.



**Fig. 3. Resulting network after removal of node 1 and transferal of leadership to node 2 in the network shown in Fig. 1.**

To define a way for our model to make these changes on its own, we can write a function that creates a new network with leadership handed down to the second-in-command. We have dubbed this function **next\_leader**. The **next\_leader** function creates a new network object with one less node, the second-in-command of the original network as the leader, and a new group of edges defined by another function named **change\_node**.

For example, suppose we want to define a new network **net\_2** by applying **next\_leader** to **net\_1**. We can then apply Coq’s evaluation functions to **net\_2** and verify that it has the properties expected of the network shown in Fig. 3. As you can see in Table 2, Coq identifies node 2 as the leader, 3 as the second-in-command, 6 as the total number of nodes, and the five edges of the network in Fig. 3.

**Table 2. Coq operations applied to object net\_2, defined as (next leader net\_1).**

|  |  |
| --- | --- |
| Coq Input | Coq Output |
| Compute leader net\_2.  Compute nodes net\_2.  Compute second\_in\_command net\_2.  Compute superior net\_2. | = 2 : nat  = 6 : nat  = 3 : nat  = [(2, 3); (2, 4); (2, 5); (3, 6); (3, 7)]  : list (nat \* nat) |

We can proceed to define more networks by applying this or any other rearrangement function to net\_1 or net\_2. More functions like this one can easily be defined by following the same principles used for this one. For example, we could expand this succession function into one that replaces any given node in a network (rather than just the leader) with its highest-ranking subordinate; we could define a function that transfers all the subordinates of node A to node B, or one that simply adds a new node as a subordinate to one already in the network.

One thing that should be noted when we remove nodes from a network like this is that we need to assert that the resulting network still has a minimum of two nodes and one edge connecting them. A single node is not a valid network as it has no edges and no way to designate a node as second-in-command.

1. Issuing Commands

We have talked about network hierarchy and reorganization in our model, but have yet to cover arguably the most important part of a command and control system, which is the commands themselves. A command, as we define here, is an instruction given by a node to its subordinate(s) telling them what to do. One way we can handle commands is to assign a numeral value to each node that indicates what it is currently doing.

Returning to our structure definition, take a look at the state field, which is a list of pairs of natural numbers. Each pair in this list contains the number of a node and another number representing its current state. For example, suppose that we want our network to have the initial state of all its nodes as “idle”. We can choose the value 1 to represent this state and define the network as follows.

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

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

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

Now, we can establish commands that change the current state of one or more nodes. Let us define the value 2 as representing the state “move”. Suppose that we want node 3 to order all of its direct subordinates to move. All we need to do is establish a way to change the state of every node that is a subordinate of node 3 to “2”.

1. Conclusion

We hope that the examples of structures, functions and properties described here can be of assistance to Coq developers in search of a general model for a command and control system or any system in which the concept of hierarchy may be relevant, as well as developers simply seeking some insight into how Coq operates.

We also intend to continue development of this model where possible by expanding it to include more complex functions and properties, particularly ones based on the ones already established here.

References

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

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

Coq Reference Manual, https://coq.inria.fr/doc

Source code, http://github.com/GGFSilva/CommandAndControl