Description of Command and Control Networks in Coq*

Guilherme G. F. da Silva[1], **Edward Hermann Haeusler**[1], **Cláudia Nalon**[2]

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

[2]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 a group of individuals organised 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 utilises Coq's implementation of data structures and includes examples of how to define functions and properties that may be relevant in a C2 system, as well as examples of some of the tests we have performed to verify the correctness and usability of our model. The examples given here are simple, but constitute basic blocks that can later be used in more realistic formalisations.*

1. Introduction

The NATO terminology database [Nato Terminology Office 2021] defines *command* as: 'The authority vested in an individual of the armed forces for the direction, coordination, and control of military forces', and *control* as: 'The authority exercised by a commander over part of the activities of subordinate organisations, or other organisations not normally under his command, that encompasses the responsibility for implementing orders or directives.' A Command and Control (C2) network harmoniously integrates both concepts with the aim of defining ways to successfully accomplish a mission. The network is often organised as an adaptive hierarchical team of agents (or coalitions) whose members collaborate over ever changing situations, applying common tactics and protocols in order to achieve some desired outcome. Not surprisingly, those networks have been widely used in military organisations as a way to minimise occurrences of unforeseen or emerging properties within a complex system [Development, Concepts and Doctrine Centre 2017]. The validation of C2 networks is, therefore, crucial not only for ensuring the full realisation of strategic goals, but also to give warranties that the accomplishment of a mission is not costly; particularly, it aims at reducing risks and minimising the loss of lives.

There are some reports in the literature on C2 formal validation. In general they fall into two approaches. One is to formalise the network as a model and to verify dynamic properties using a temporal logic model-checker; the other is to formally prove relevant properties, using an Interactive Theorem-Prover (ITP), or even an Automatic Theorem-Prover (ATP). For instance, [Wang 2012, Chapter 8] describes the use of stochastic Petri-Nets to formalise the validation of C2 networks applying the first

*This work was partially funded by the project FINEP 2904/20 - Sistema de Sistemas de Comando e Controle (S2C2).

approach. [van de Pol et al. 1998] proves some very basic properties using the interactive prover PVS [Owre 2020] together with a model-checker embedded in that system to validate models generated from the specification. The latter seems to be one of the first works integrating both approaches, but it is restricted to the validations of instances/configurations of particular C2 networks. Due to the widespread popularity of model-checking and its apparently easier end-user modelling, there are more reported approaches using model-checkers than logical formalisations, or even about the formal mathematics of C2. We however believe that the theorem-proving approach is better than model-checking to ensure the absence of non-normative features, as required in the validation of C2 networks.

This work is part of a project that aims to formally develop and validate a methodology to ensure that C2 networks are correct with regard to non-emerging properties as much as possible. As far as we know, this is the first project that is based on theorem-proving with the focus on normative specification of C2. We started the formalisation using Coq [The Coq Team 2021] and the very basic blocks of C2 networks. This article reports the achievements obtained for these building blocks of C2.

2. Representation of a Network in Coq

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. A valid network has at least one node. 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 the relation between individuals and their direct subordinates.

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, and a network must possess a minimum of two nodes and one edge. Figure 1 shows an example of a graph representing such a system, where individual 1 is the leader, 2 and 3 are her direct subordinates, and so on.

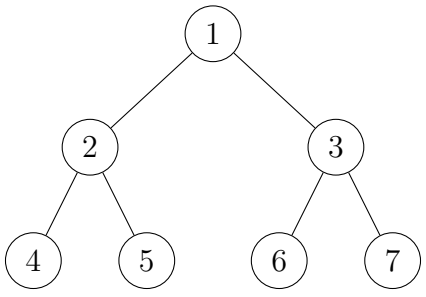

Figure 1. Directed graph representing the hierarchy in a command and control network

2.1. 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, which can be found at [Silva, Guilherme 2021]. This net object represents a command and control network as a structure type containing a single leader node and a set of nodes subordinated to this leader. These subordinates, in turn, can have their own subordinates, and so on. The objects and functions that make up this structure are as follows.

We define the number of nodes in our model as a variable named `nodes`, which is a 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. The largest network we have used when testing our model has a total of 31 nodes.

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. 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 Figure 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 two 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.

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.

We detail the implementation of these functions in the next section.

2.2. 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**. This 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 this function does is recursively search through the list of edges (a, b), 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 (a, b). 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. 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 take two nodes a and b as input and return whether a is a parent of b. Coq has two different types to represent this, proposition (Prop) or boolean (bool). Prop defines the constants for truth and falsity in the metalanguage whereas bool is a defined type similar to the one used in programming languages. For comparison's sake, we have included those two different functions, one for each of these types. The two are mostly similar but with some differences in which operators are used. 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. The recursion stops at the node representing the leader. 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 $O(n^2)$ 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 [Silva, Guilherme 2021].

2.3. 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. In Figure 2, we revisit the network example shown earlier in Figure 1, and we can see that it has three different levels of hierarchy.

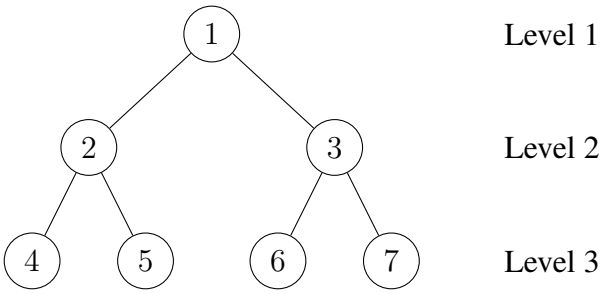

Figure 2. Example of a network with seven individuals and three hierarchy levels.

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.

Coq Input	Coq Output
Compute is_parent_bool `net_1` 1 2.	= true : bool
Compute is_parent_bool `net_1` 2 3.	= false : bool
Compute node_level `net_1` 1.	= 1 : nat
Compute node_level `net_1` 2.	= 2 : nat
Compute node_level `net_1` 3.	= 2 : nat
Compute node_level `net_1` 4.	= 3 : nat
Compute parent `net_1` 2.	= 1 : nat
Compute children `net_1` 1.	= [2; 3] : list nat

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

2.4. 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. We exploit properties like these 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)).
```

These are a few examples of basic properties that any network in our model must have. Properties like these can be used as a stepping stone for defining more complex properties as well as for constructing theorems and proofs.

3. 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 organisation 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 Figures 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 2.2, node 2 is the one considered this network's second-in-command. So let us consider what might happen if node 1 were removed 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. Figure 3 shows the resulting network that we expect from this transformation.

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

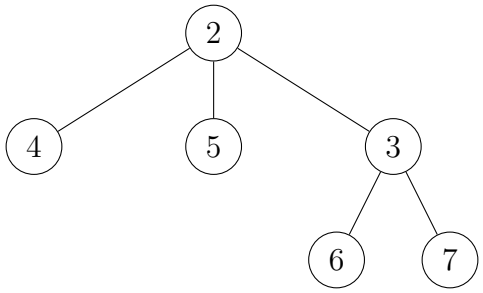

Figure 3. Resulting network after removal of node 1 and transferal of leadership to node 2 in the network initially shown in figure 1.

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 Figure 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 Figure 3 .

Coq Input	Coq Output
Compute leader `net_2`.	= 2 : nat
Compute nodes `net_2`.	= 6 : nat
Compute second_in_command `net_2`.	= 3 : nat
Compute superior `net_2`.	= [(2, 3); (2, 4); (2, 5); (3, 6); (3, 7)] : list (nat * nat)

Table 2. Coq operations applied to object `net_2`, defined as (next leader `net_1`.)

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.

4. Issuing Commands

We have talked about network hierarchy and reorganisation 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".

5. 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 the 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

Development, Concepts and Doctrine Centre (2017). Future of command and control. *Joint Concept Notes*, 17(2).

Nato Terminology Office (2021). Natoterm. `https://nso.nato.int/natoterm/content/nato/pages/home.html?lg=en`. Last visited: 17-Sep-2021.

Owre, S. (2020). PVS 7.1, The Prototype Verification System. `https://pvs.csl.sri.com/`.

van de Pol, J., Hooman, J., and Jong, E. (1998). Formal requirements specification for command and control systems. In *In Proceedings of the Conference on Engineering of Computer Based Systems*, pages 37–44. IEEE.

Silva, Guilherme (2021). Source code. `http://github.com/GGFSilva/CommandAndControl/`.

The Coq Team (2021). Coq 8.13. `https://coq.inria.fr/`.

Wang, J. (2012). *Timed Petri Nets: Theory and Application*. The International Series on Discrete Event Dynamic Systems. Springer US.