Verification of an Airport Taxiway Path-Finding Algorithm

Siyuan He
School of Information
University of Michigan
Ann Arbor, MI, USA
hesy@umich.edu

Ke Du
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI, USA
madoka@umich.edu

Joshua Wilhelm
Aerospace Engineering
University of Michigan
Ann Arbor, MI, USA
joshwilh@umich.edu

Jean-Baptiste Jeannin
Aerospace Engineering
University of Michigan
Ann Arbor, MI, USA
jeannin@umich.edu

***Abstract*—At controlled airports, aircraft taxi on the ground following precise instructions from Air Traffic Control. Those instructions can get quite intricate and complicated and lead to errors from Air Traffic Control or confusion from pilots, especially at larger airports. In order to reduce the pilots' workload and lower the risk of accidents from pilot error or Air Traffic Control mistakes, it is desirable to automate taxiing of aircraft.**

One aspect of automated taxiing is to automatically find the correct taxiway path from Air Traffic Control instructions. In this paper, we implement and formally verify a taxiway path-finding algorithm in the Coq proof assistant, and test it on three different airports of various sizes (Ann Arbor, Willow Run and Detroit Wayne). We first build an undirected airport taxiway graph topology, extend it to a directed expanded graph, disallowing some unrealistic paths such as U-turns, and formally encode the two graphs into Coq types. We then implement the path-finding algorithm on the directed expanded graph, and map downward the result back to the undirected graph. We formally verify the correctness of our algorithm in the Coq theorem prover.

***Index Terms*—verification, proof assistant, graph algorithms, airport taxiway**

I. Introduction

Automation has become widespread in commercial aircraft over the past decades, but automation in ground operations has not been widely implemented yet. The taxi instructions provided to pilots by Air Traffic Control through radio voice commands can lead to confusion, especially at major airports with many taxiways. A pilot error could lead to taking the wrong taxiway or crossing a runway without authorization, leading to potential accidents. At the same time, Air Traffic Control can make mistakes, and pilots must ensure that the instructions correspond to a valid path on the airport taxiways (i.e., the instructions are accurate) and that this path is unique (i.e., the instructions are not ambiguous). If not, a pilot would get back to Air Traffic Control requesting corrected instructions.

In order to reduce the pilots' workload on the ground, let them focus on their flight, and reduce potential accidents, it is desirable to automate taxiing of aircraft. Part of this task is to generate a valid taxiway path from Air Traffic Control instructions. Since the correctness of this path is safety-critical, we would also like to verify that this path is correct with respect to the instructions that were received.

The problem of avoiding runway incursions is the object of many papers, including several surveys [1], [2]. Cheng et al. [3] identify the importance and potential airport efficiency improvements of an autonomous taxiing system. Liu and Ferrari [4] present a vision-based self-taxiing approach in which the airport is also modeled as a graph, similar to our method. While their path-finding algorithm is similar to ours, they do not attempt to formally verify it. On the other hand, Fremont et al. [5] present a vision-based centerline-tracking autonomous taxiing system and perform its formal analysis. However, they focus on staying on a pre-determined taxiway and do not deal with the path-finding step of determining the correct taxi route. Julian et al. [6] investigate the problem of validating vision-based neural network controllers. For these types of controllers, formal verification is very difficult, so they propose an alternative consisting of a combination of adaptive stress testing and neural network verification tools. Additionally, Lu et al. [7] present a self-learning approach to autonomous taxiing, and Eaton and Chen [8] present an image-based collision detection system for autonomous taxiing. Both of these papers focus on following taxiways or avoiding collisions using vision.

Formal verification can be used to ensure that an aerospace system meets its specifications. This is important, as attempting to find errors in a system or software through standard processes becomes more difficult as system complexity increases. There are many examples of successful use of formal methods to verify various aerospace systems, e.g., [9]–[12]. Formal verification on graphs or graph-based algorithms has been investigated using different techniques, e.g., da Costa and Ribeiro [13] and Kupferman [14].

In this paper, we use the Coq proof assistant [15] to formally verify the correctness of an airport taxiway path-finding algorithm. The algorithm takes as inputs an undirected graph representing the map of the airport, start and end locations represented as vertices of the graph, and Air Traffic Control instructions consisting of a list of taxiway names. It outputs a unique valid path, or an error if the path is either nonexistent or not unique. At its heart, the path-finding algorithm is based on a breadth-first search, modified and optimized to ensure that we only follow valid paths corresponding to the instructions. The formal verification is performed in the Coq theorem prover, and consists of 2,400 lines of code including proofs.[1]

II. Graph Encoding

A natural way to represent an airport ground layout is to use an *undirected* graph G, setting the edges as the taxiway branches and the vertices as the intersections. We follow the naming convention described by Zhang et al. [16] for the undirected graph. We use the names of taxiways to name the vertices at the intersections of taxiways.

Let us take the Ann Arbor airport (KARB) as an example. From its official airport taxi diagram (Fig. 1), we extract an undirected graph (Fig. 2 [16, Fig. 3]), where vertex AB is named from taxiway A and taxiway B (we do not consider $A2$ which is a minor taxiway). We can simply represent the edge as a pair of two vertices, for example $(AB,\ AC)$ means the edge between AB and AC.

One problem with the undirected graph representation is that it allows illegal paths such as U-turns, which aircraft cannot take in reality. In other words, the algorithm does not remember from which taxiway the aircraft reached its current position. For example in Fig. 2 [16, Fig. 3], if the aircraft is currently at vertex AC and is ordered to go to AB by path-finding on the undirected graph, we cannot know whether it came from $AA1$ or BC. If it came from BC, some large aircraft may not be able to make the tight turn toward AB. For this reason, we expand the undirected graph into a *directed expanded* graph. The directed expanded graph is similar to a line graph [18], but with some slight modifications.

To avoid confusion, we refer to the elements of the undirected graph as *vertex* and *edge*, while we use *node* and *arc* for the directed expanded graph. A node in the directed expanded graph encodes the current vertex in the undirected graph along with the last vertex the aircraft came from. Thus, a node in the directed expanded graph corresponds to an edge in the undirected graph. Since the arc is directed in the directed expanded graph, we use an ordered pair of two nodes to represent an arc. For example, the arc (x, y) means "from node x to node y". Note that the node (x, y) means "from y to x", but arcs are encoded in the inverted order. Thus the arc $((AC,\ BC),\ (AB,\ AC))$ means that the aircraft came from BC, is currently located at AC, and is moving to AB. For clarity we denote such an arc as $((AC \text{ from } BC) \text{ to } (AB \text{ from } AC))$. Intuitively, an arc is an ordered pair of $(previous\_edge,\ current\_edge)$, and it becomes an edge in the undirected graph if we drop the first term. With the help of the directed expanded graph, we are able to drop paths that are illegal in reality [16].

In our type definition, we further encode the taxiway name into edges and arcs. The modified definition encodes the taxiway names into each edge or arc, so we do not need to search for the taxiway name in a global record each time. We construct a pair with the first element as the original edge or arc and the second element as the taxiway name. Formally, the *Edge_type* requires a pair of two vertices and a taxiway name, so an instance can be $((Ch,\ BC),\ C)$ where $(Ch,\ BC)$ is the original edge, and C is its taxiway name.

Formally, the type definition starts from two basic types, *Vertex* representing vertices in the undirected graph and *Taxiway_type* representing taxiway names. The rest of the types are built through pairs of basic types or constructors of basic types. The types of the directed expanded graph are encoded in the Coq proof assistant as:

```
  Node_Type := Vertex * Vertex
   Arc_Type := (Node_Type * Node_Type) * Taxiway_Type
C_Graph_Type := list Arc_Type
```

Note that we encode a graph as an ordered list of *edges* or *arcs*, but allow repetition and ignore ordering so it represents a set, because we only care about the inclusion relation of a graph and its elements. A path in a graph is also a list of *edges* or *arcs*, but adding repetition and reordering will make it a different path, therefore we view it as an ordered list as encoded.

Since we assign every component in the undirected or directed expanded graphs a different type, the strong type system of the Coq proof assistant ensures that we always pass the right component to functions in our implementation.

In real airports, the taxiing path can only start from certain places, which are represented by double circles in Fig. 2. Therefore we need to restrict input vertices and nodes in our graph. We have a special instance of the $Vertex$ type, the "*input*" vertex, in our design. We hardcode the "*input*" vertices and add a unique extra edge pointing to legal input vertices in the original undirected graph to identify input vertices. For example, the pair $((Ch,\ input),\ \text{""})$ is an edge instance in the undirected graph saying that Ch is a input vertex. After conversion to the directed expanded graph, we are able to enforce the next arc to come from this extra *input edge*, such as $(((Ch,\ input),\ (BC,\ Ch)),\ C)$.

III. Algorithm

We divide the path finding algorithm into three parts. Our algorithm is inspired by previous work [16] but modified in key places to ease verification. The first part is an expanding algorithm that turns an undirected graph into its associated directed expanded graph. The second part finds all possible paths on the directed expanded graph. Finally, we call a downward mapping algorithm to turn the path in the directed expanded graph to one in the undirected graph. One can additionally modify the output of the expanding algorithm from step 1 to specify all the paths that an aircraft can take before executing the second part. The correctness properties will still hold as long as the modified directed expanded graph meets its specification, given in Section IV.

Although a graph and a path bear different properties (the former is viewed the same after adding repetition or reordering), our expanding and downward mapping algorithms preserve ordering, so the input of these algorithms can be both a graph and a path. This design is validated by the proof.

[1]Proofs are available at
https://github.com/rinshankaihou/pathway_finding_verification

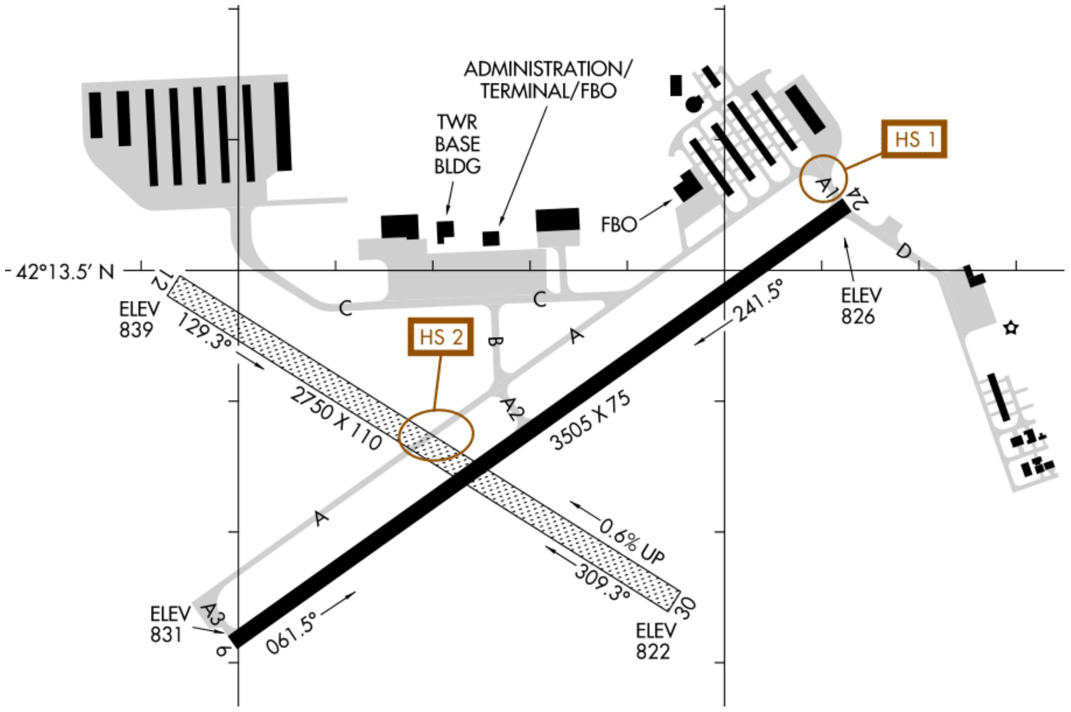

Fig. 1. Official Ann Arbor airport taxi diagram published by the Federal Aviation Administration [17]

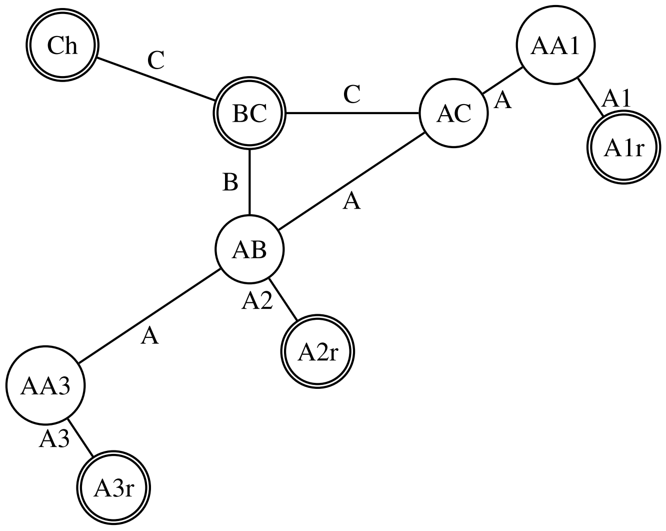

Fig. 2. undirected graph of Ann Arbor airport (KARB) from [16]

Before we step into the details of our algorithm, we first introduce some symbol conventions we will use in the rest parts of our paper, in the order of decreasing precedence.

$s@n$, the n^{th} element of a tuple s,
including the state that will be introduced below

$[a]$, a list containing only one element a

$h :: l$, a list constructed by appending h to the head of l

$l_1{+}{+}l_2$, a list constructed by concatenating l_2 at the end of l_1

A. From Undirected to Directed Expanded

As stated above, the undirected graph has limitations when dealing with illegal paths, so we start by expanding the undirected graph to a directed expanded graph to allow encoding of more constraints in the graph.

These constraints forbid, or filter out certain illegal paths for an aircraft in the directed expanded graph, such as paths containing a U-turn, which the undirected graph is unable to encode. In our verified implementation the filter is an identity map; i.e. the graph is not filtered. However we expect an arbitrarily customized filter to preserve the correctness, as long as the filter outputs a subset of the directed expanded graph.

```
DIRECTED-EXPANDED-GRAPH($UG$)
    Data: undirected graph $UG = (\_V, E)$
    Result: directed expanded graph $DEG = (\_N, A)$
    create $DEG.A = [\,]$
    expand $UG.E$ to bi-directional graph $E'$
    foreach edge $e \in E'$ do
        prev = [$e' \in E'$, $e$.from= $e'$.to and $e'$.to$\neq$ $input$]
    end
    drop $e'$ $\in$prev if $e'$ =inv $e$
    foreach directed edge $e'$ $\in$prev do
        append (($e'$.fst, $e$.fst), $e$.taxiway) to $DEG.A$
    end
    return $DEG$
```

Algorithm 1: Generation of directed expanded graph (inspired from [16])

The undirected graph is encoded as a list of *ordered* vertex couples, along with the name of the corresponding taxiway, such as $((AC, AB), A)$ (Fig. 3), but the order does not matter at this point. The undirected graph of Ann Arbor airport is given in Fig. 2, and its directed expanded graph is given in Fig. 5.

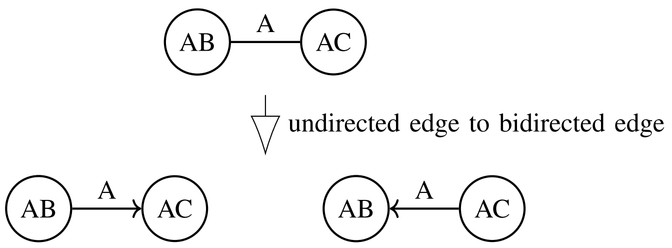

Fig. 3. A mapping from an undirected edge to bidirected edges.

The expansion algorithm has two steps. It starts by turning each undirected edge into two directed edges. A directed edge has the same encoding, except the order matters in our algorithm. For example, in Fig. 3, $((AC, AB), A)$ is an undirected edge between AB and AC on taxiway A, which is expanded to $(AC \text{ from } AB)$ and $(AB \text{ from } AC)$, both on taxiway A. Applying such an operation gives us the *bidirected graph*. Since an airplane is not allowed to go to the "*input*" vertex, we additionally filter out the edges with the form "$((\_,\ input),\ \_)$" in the bidirected graph.

Then for each directed edge in the bidirected graph, an edge along with one of its previous edges describes a possible path. Packing an edge, the taxiway name of that edge, and a previous edge gives us an expanded *arc*. All the possible *arc*s constitute the directed expanded graph. In Fig. 4, the directed edge $(AC \text{ from } AB)$ has 3 previous edges, which results in 3 *arcs*: $(((AB \text{ from } BC) \text{ to } (AC \text{ from } AB)), A)$, etc. For each edge in the bidirected graph, we find all possible previous edges from the bidirected graph and construct the form. Since the aircraft will never turn around and go back on the same edge, we forbid the previous edge to be the inverse of current edge, so an arc such as $(((AB \text{ from } AC) \text{ to } (AC \text{ from } AB)), A)$ will not appear.

We complete the generation of the directed expanded graph by repeating the operation described in Fig. 4 for every bidirected edge. If we apply the expanding algorithm (Algorithm 1) to the Ann Arbor airport (Fig. 2), we get the directed expanded graph as shown in Fig. 5.

Note that the idea of the expansion algorithm comes from the directed expanded graph generation algorithm discussed from [16, Algorithm 1]. However, the detailed design of our Algorithm 1 is slightly different from theirs. The modification results in shorter and clearer code in Coq and an easier proof.

B. Path-finding on Directed Expanded Graph

The algorithm find-path (Algorithm 2) takes a directed expanded graph and an Air Traffic Control (ATC) command as input, and outputs a list of all possible paths. Note that the input graph is not necessarily the full graph generated by the

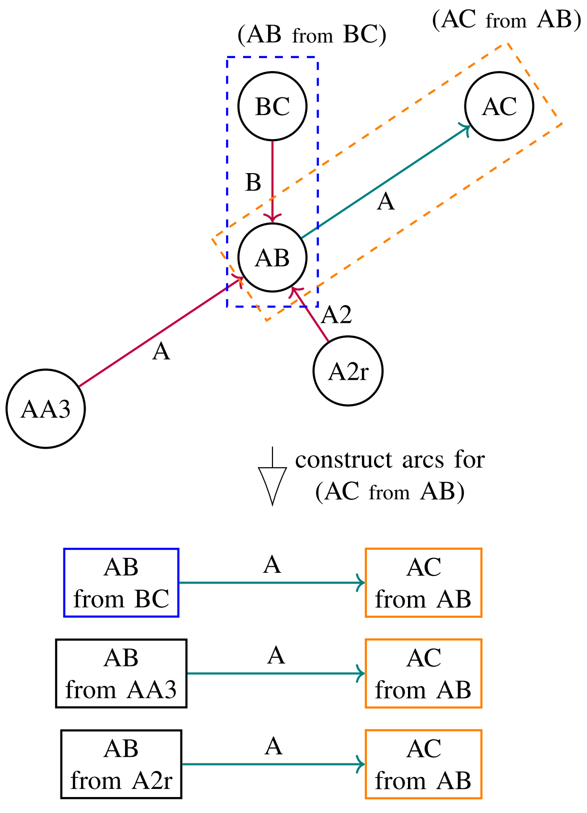

Fig. 4. Expansion of an edge to an arc

expansion algorithm (Algorithm 1), as we can customize the graph by filtering out illegal arcs. The filtering algorithm can be arbitrarily customized, as long as the output is a subset of the input graph.

The design of Algorithm 2 is again inspired from the *Path Finding: Auxiliary Recursive Algorithm* proposed by [16, Algorithm 2]. The idea is based on breadth-first search, where in each iteration we step an arc further from the starting arc. The modified algorithm is easier to implement in a functional language such as Coq.

We introduce the structure *state* of *State_type* to pack all necessary information of a search state into one instance, and all intermediate operations will be conducted on *states* in the algorithm find-path.

$$\begin{aligned}\texttt{State\_type} := \mid \texttt{State}:\ &(\texttt{list Arc\_type}) \to \texttt{string} \to \\ &(\texttt{list string}) \to (\texttt{list string}) \\ &\to \texttt{State\_type}\end{aligned}$$

A state should be constructed through the constructor `State` along with four arguments, so it is essentially a 4-tuple. The four arguments are

$s@1$	the list of history arcs we have come through to the current position
$s@2$	the taxiway name of the current position
$s@3$	the remaining ATC commands we will deal with
$s@4$	the ATC commands we have used

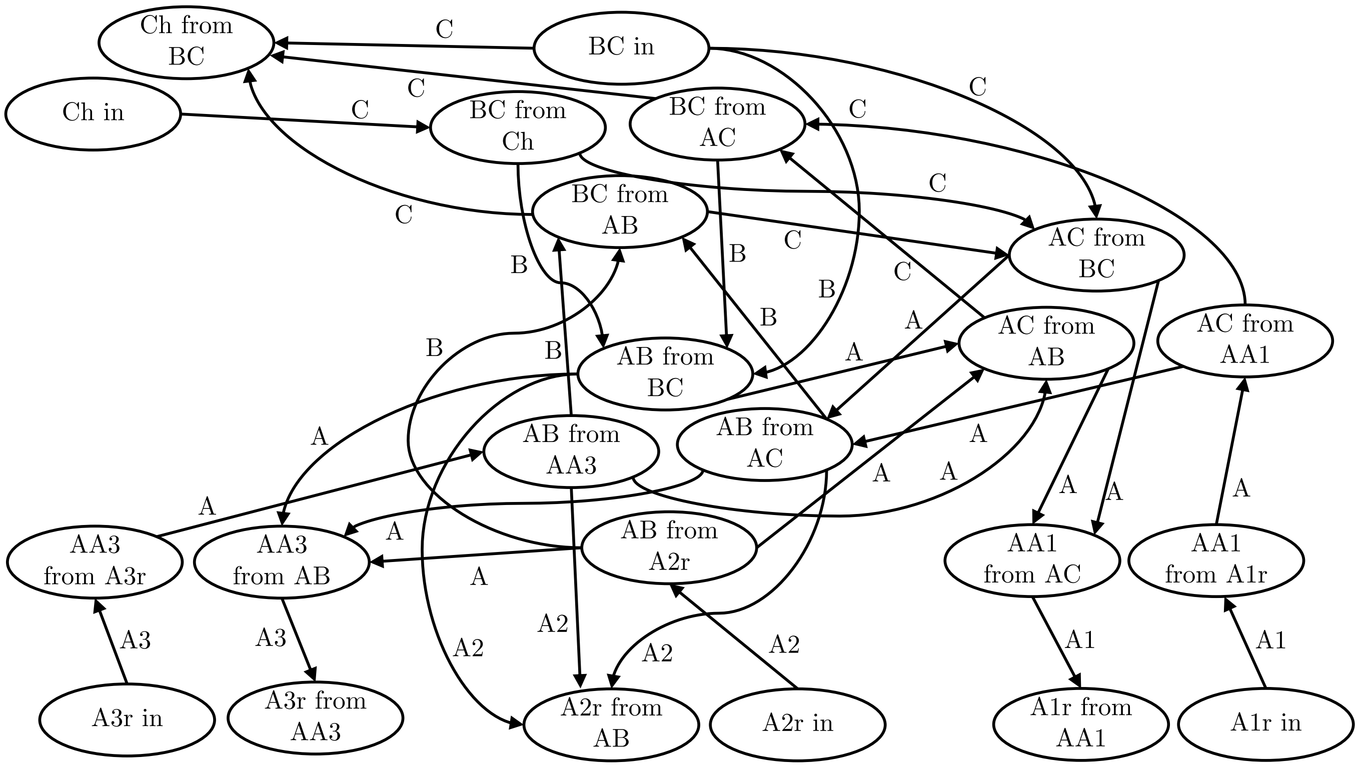

Fig. 5. Directed expanded graph of the taxiways at Ann Arbor airport (KARB) [16]

Note that the history arcs and the used ATC commands are organized in reverse order in the list, because we append the current arc/ATC command to the head of the list during each step. An invariant of the state is that the initial ATC command can always be recovered at any point. Formally, we write the property as $(\text{rev } s@4) ++ [s@2] ++ s@3 = \text{ATC input}$, meaning that the three parts in the state represent the used, current, and future taxiway names in the ATC command. We maintain the invariant by either doing nothing or taking a new taxiway as current taxiway and appending the last current taxiway to the history taxiways.

In the find-path algorithm (Algorithm 2), we start by packing the start vertex and ATC command into an initial state and inserting the initial state into a queue. Then we repeat popping and handling the first element of the queue until reaching a pre-set maximal round or until the queue is empty. For each state, we first filter all arcs in the graph that start from the current position, and then check whether the taxiway name of each remaining arc is the same as the current taxiway or the next taxiway in the ATC command. Either condition means we can go one step further on the arc, so we create a new state keeping the invariant, and push the new state into the queue.

Note that we have a pre-set bound for maximal steps `bound`. The bound for maximal steps is only used to explicitly ensure that the program will terminate. The aim of introducing the bound for maximal steps is to make it easier to prove correctness, which will be discussed later.

C. Downward Map back to Undirected Graph

The find-path algorithm (Algorithm 2) returns a list of paths in the directed expanded graph, and each path is a list of *arcs*. In order to have a more natural representation, we map a path consisting of arcs to a path with edges. Luckily, an arc contains every necessary piece of information to reform an edge, and we call the map from `Arc_type` to `Edge_type` a "downward map".

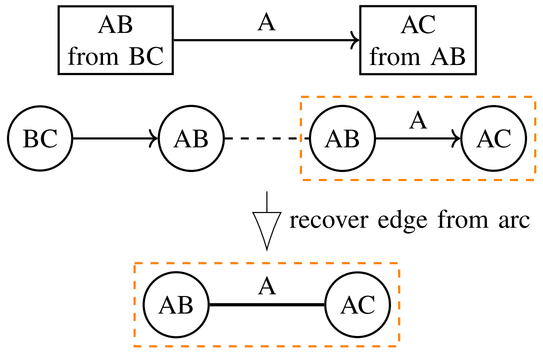

Fig. 6. Downward Map from an Arc to an Edge

In the expansion algorithm (Algorithm 1), we attach a previous edge to an edge to form an arc. Hence simply dropping the previous edge recovers the edge. For example, in Fig. 6, dropping the first half of the arc $(((AB \text{ from } BC) \text{ to } (AC \text{ from } AB)), A)$ gives the original edge $((AC \text{ from } AB), A)$. Applying this operation to all arcs in the

```
FIND-PATH($DEG$, ATC, start_v, end_v)
  Data: directed expanded graph $DEG = (\_N,\ A : list\ Arc\_type)$, ATC command ATC: $list\ Taxiway\_type$, start
        vertex start_v: $Vertex$, end vertex end_v: $Vertex$
  Result: a list of all possible paths DEp: $list\ list\ Arc\_type$
  create DEp= [ ]
  set a proper bound for maximal steps bound, default is 100
  set the initial state init=State [(((start_v, input), (start_v, input)), head(ATC))] head(ATC) rest(ATC) [ ]
  create queue= [init]
  for 1 to bound do
    foreach $s$ ∈queue when the iteration starts do
      pop $s$ from queue
      // check if endpoint reached
      if head(s@1).to_node.to =end_v && length(s@3) = 0 then
        append (rev $s$@1) to DEp
      end
      // find all arcs starting from current position
      create next_arcs= [$a \in DEG.A$ if $a$.from_node = head($s$@1).to_node ]
      // pack arcs with correct taxiway name into a state and append to queue
      foreach $a$ ∈next_arcs do
        // next arc is on the current taxiway in ATC command
        if $a$.taxiway = s@2 then
          push (State ($a$ :: $s$@1) $s$@2 $s$@3 $s$@4) to queue
        end
        // next arc is on the next taxiway in ATC command
        if $s$@3 $\neq$ [ ] && $a$.taxiway = head(s@3) then
          push (State ($a$ :: $s$@1) $s$@2 (tail($s$@3)) ($s$@2 :: $s$@4)) to queue
        end
      end
    end
  end
  return DEp
```

Algorithm 2: Find-path on directed expanded graph (modified from [16])

path gives a path of edges, which is described in Algorithm 3. This is also the final output of the entire path-finding algorithm.

```
DOWNWARD-MAP(DEp)
  Data: a list of paths DEp : $list\ list\ Arc\_type$
  Result: a list of paths Up: $list\ list\ Edge\_type$
  create Up= [ ]
  foreach $al$ ∈DEp do
    create $p$ = [ ]
    foreach $a \in al$ do
      append ($a$.to_node, $a$.taxiway) to p
    end
    append $p$ to Up
  end
  return Up
```

Algorithm 3: Downward map from arcs to edges

As stated before, the expansion and downward-map algorithms work on both graphs and paths, so we do not distinguish the input, and we denote the expansion function as `to_C` (as in to_complete_representation), downward-map function as `to_N` (as in to_naive_representation). Then the path finding algorithm can be described as

$$\texttt{to\_N} \circ \texttt{find-path} \circ \texttt{to\_C}$$

.

IV. Correctness

At the highest level, the correctness of our algorithm means that for every path we find, the output path is a correct path for the input. Since our algorithm is a combination of three separate parts, we modularize our proof accordingly. We first prove the correctness of the find-path algorithm (Algorithm 2) on an arbitrary directed expanded graph, then we prove that the correctness is preserved with the downward-map algorithm (Algorithm 3) from arcs to edges, i.e. `to_N` ∘ `find-path` is correct. Lastly we prove the entire algorithm is correct. We further prove the expansion algorithm (Algorithm 1) and the downward-map algorithm (Algorithm 3) are correct, i.e. `to_C` ∘ `to_N` is an identity map under certain conditions, therefore it behaves very similar to one's intuition. The last property complements the correctness definition because otherwise, some absurd implementations of the expansion algorithm (Algorithm 1) and the downward-map algorithm (Algorithm 3) may still make the entire algorithm satisfy our correctness definition. This will be discussed in detail.

As shown in the proof logic illustration (Fig. 7), we ultimately want to prove that the undirected path "path(edge)" we find is correct given the "undirected graph". Intuitively, it is hard to define the correctness of the expanding algorithm (Algorithm 1) and determine how the correctness can contribute to the correctness of the combined algorithm, hence we start with the correctness of find-path algorithm (Algorithm 2) on the directed expanded graph.

Instead of taking the expanding algorithm (Algorithm 1) into consideration at this point, we prove the correctness of find-path to hold for not only the output of `to_N` but all directed expanded graphs. In Fig. 7, the correctness for find-path algorithm (Algorithm 2) states that "path(arc)" is correct given some "directed expanded graph". Then we prove the downward preservation property of the correctness when the downward map algorithm (Algorithm 3) is applied to both the path and the corresponding graph: the correctness of the find-path algorithm (Algorithm 2) is preserved by the downward map algorithm, i.e., "path(arc)" is correct on "directed expanded graph" indicates "path(edge)" is correct on "UG'" in Fig. 7. Note that we can only determine whether a path is correct based on the exact graph input to the algorithm. Hence, the theorem that the "path(edge)" is correct on "UG'" is not strong enough to support the correctness of the theorem that "path(edge)" is correct on "undirected graph". The goal now is changed to showing that a path is correct on the graph created by applying the downward map (Algorithm 3) after applying the expanding map (Algorithm 1) to some undirected graph indicates the path is correct on the undirected graph. Formally,

$$\begin{aligned}&\forall \ \ p\ G,\\ &\quad \texttt{correct}\ p\ (\texttt{to\_N}\ (\texttt{to\_C}\ G)) \rightarrow \texttt{correct}\ p\ G\end{aligned}$$

where `to_N` and `to_C` represent the downward map (Algorithm 3) and the expanding map (Algorithm 1), respectively. Alternatively, we focus on the identity of the two maps to prove the goal. The identity describes the property that if an edge is either in a graph G (under some reasonable specification) or ($\texttt{to\_N}$ ($\texttt{to\_C}$ G)), then the edge should also be in the other graph. By the correctness of find-path in Sec. IV-A, correctness preservation in Sec. IV-B, and identity of two maps in Sec. IV-D, we finish the proof for the correctness at the highest level that "path(edge)" is correct on "undirected graph" as illustrated in Fig. 7.

We will see that the correctness of the find-path algorithm (Algorithm 2) does not depend on the input graph, so find-path on a filtered graph is trivially correct, and although this is left as future work, we expect the correctness to be trivially preserved with any kind of filter. The correctness of our algorithm is discussed under the condition that the filter is an identity map as shown in Fig. 7, so in the following sections the filter can be safely ignored.

A. Find-path Correctness

The first part of the correctness proof states that for every path with *list Arc_type*, the find-path algorithm (Algorithm 2) returns, the path is a correct path based on the directed expanded graph. We claim that a path is correct if it satisfies:

- The path starts at the correct node indicated by `start_v`.
- The path ends at the correct node indicated by `end_v`.
- Every arc in the path is an arc in the directed expanded graph.
- The path is a connected path, i.e. the "*to_node*" of an arc in the path is always equal to the "*from_node*" of the arc of the next position in the path.
- The path follows the ATC command. The taxiway names of the arcs should be of the form $[\text{ATC}_1^+ \text{ATC}_2^+ \text{ATC}_3^+ \ldots]$, where ATC_i is the i^{th} element in the ATC command, and a $^+$ superscript refers to being repeated one or more times as in regular expressions.

The intuitive way to prove this part is by induction on the bound for maximal steps counting down to zero in the find-path Algorithm 2. However, it is unwise to prove by induction on the bound for maximal steps because our algorithm will not convert a state to a path and add to the result list until the state reaches the endpoint. If there is a new path added to the result list in exactly the `bound` round, the induction hypothesis is useless to prove that the new path is also correct.

Alternatively, we prove by the invariant of states. The state invariant can be derived from the algorithm in Sec. III-B, which describes the property of a state s generated by s' in an iteration where:

- $s@1$ inherits all arcs from $s'@1$ and has one more arc than $s'@1$, i.e. $s@1 = a :: s'@1$ for the a point to current position.
- (rev $s@4$) $++$ $[s@2]$ $++$ $s@3$ is always equal to the input ATC command, where (rev $s@4$) is $s@4$, a list of the used ATC command, in reversed order.

The intermediate data structure used in the find-path algorithm is the *State_type*, and in each iteration we pop some states and append some new states. In our implementation, we call the function "*step_states*" to generate new states, stepping one edge farther from the current state. The state invariant ensures that some properties of the new states generated (by "*step_states*") in this iteration are correct if the properties hold for the given state. In other words, the properties of a state are invariant through each iteration (on "*step_states*").

With the state invariant, we can change the proof of the correctness for the output result to the proof that the correctness holds and is kept on an arbitrary state s as follows:

- The last arc (due to the reverse order) in $s@1$ is always the arc indicated by `start_v`.
  This can be proved by proving the initial state starts from the correct arc and the arc is inherited through iterations.

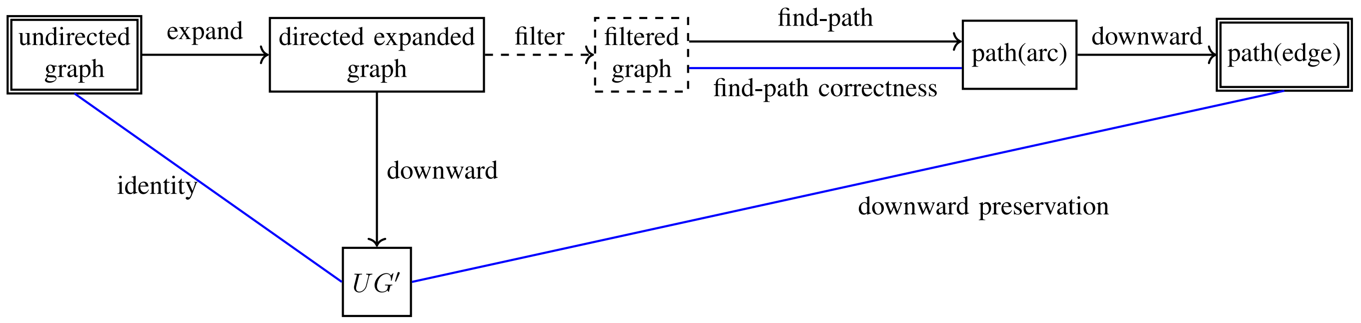

Fig. 7. Proving logic of the correctness. The black arrows mean "generate", and "UG'" is the undirected graph generated by applying the downward map algorithm (Algorithm 3) after the expanding map algorithm (Algorithm 1) to the input "undirected graph". The blue lines mean the pair of components we determine correctness on. For example, we determine whether "path(edge)" is correct based on graph "UG'", which is proved by downward preservation in Sec. IV-B. The "filter" operation is any function that generates a subset of the directed expanded graph without harming the correctness, including an identity map dropping nothing.

- The head of $s@1$ is always the arc pointing to the current position, and the last position iterated is the correct position.
  This property can be directly derived from the state invariant and the end check condition.
- Any arc a appended to $s@1$ during iterations is always in the directed expanded graph . The state invariant tells us that we append an arc to $s@1$ each time, and by hypothesis we can assume the existing arcs in $s@1$ are in the graph.
- Any arc a appended to $s@1$ during iterations is connected to the last current arc $head(s@1)$ in the graph.
  Similarly, the invariant tells us that we can assume that $s@1$ is connected, hence we prove that the connected property is kept.
- If the path to current position recorded in a state s follows the ATC command recorded in s, then for any state s' generated from s, the statement holds.
  This property is much more complicated than the other four because it involves both the path and the ATC command, so we need to use all state invariant properties. First notice that (rev $s@4$) $++s@2$ is the corresponding ATC command for the path from the start position to the current position. We want to prove that the reverse of $s@1$ (the path to current position) always follows the partial ATC command (rev $s@4$) $++s@2$, which we call "partial follow ATC". We prove the "partial follow ATC" property by state invariant and then prove that $s@3$ (unused ATC) should always be empty when the state is converted to the output path. This means we have used all of the ATC command so that the output path follows the complete ATC command.

The state invariant properties are like the inductive hypothesis on the steps we search in the graph. We generalize the bound for maximal steps so that the state invariant properties are not restricted by some specific iteration. Hereby we solve the problem raised in our attempt of induction on the bound. With the state invariant and the derived properties, we can prove the correctness of find-path easily like a normal induction.

B. *Downward Preservation*

As illustrated in Fig. 7, the downward preservation says that if the "path(arc)" is correct on the "directed expanded graph", then the "path(edge)" is correct on the graph "UG'". Formally, we can write the downward preservation as

$$\begin{aligned}&\forall\ (p : \texttt{list Arc\_type})\ \ G,\\ &\quad \texttt{correct}\ p\ G \to \texttt{correct}^*\ (\texttt{to\_N}\ p)\ (\texttt{to\_N}\ G)\end{aligned}$$

where `to_N` is the downward map. Note that we use $\texttt{correct}^*$ for downward correctness because we should rewrite the correctness theorem for `Edge_type` instead of `Arc_type` due to the type constraint of Coq. We divide downward correctness into five sub-statements corresponding to those of correctness.

Most of the properties are proved by solely investigating the invariants of Algorithm 3, except the connectedness property, which requires an additional condition that the input expanded graph fed to Algorithm 2 consists of *legal* arcs. Intuitively, a "legal" arc is one for which the head of its previous edge meets the tail of its current edge. For an arc $((V1 \text{ from } V2), (V3 \text{ from } V4)$, recall that $((V1 \text{ from } V2)$ is the previous edge, and $(V3 \text{ from } V4)$ is the current edge the airplane is on, therefore we call an arc *legal* if $V1 = V4$. In addition, we proved that the expanded graph given by Algorithm 1 contains only *legal* arcs, and that the arcs of any output from Algorithm 2 come from the input graph, therefore Algorithm 2 only outputs a *legal* path if the expanded graph is a subset of one that is generated by Algorithm 1.

C. *Correctness of path finding*

Finally, we can prove the correctness of the entire path finding algorithm.

Downward preservation, i.e. ($\texttt{to\_N} \circ \texttt{find-path}$) working correctly on any *directed expanded* graph implies it also works on ($\texttt{to\_C} \circ UG$) for some undirected graph UG. Therefore the correctness properties of ($\texttt{to\_C} \circ \texttt{find-path} \circ \texttt{to\_N} \circ UG$) are automatic if user input UG does not appear in the conclusion. There is one last property that does not fall into this category, which states that every edge in the output of the path finding algorithm comes from the input graph UG. More specifically, they come from the bidirected version of UG, because our undirected graph is a list of *directed* edges, and conceptually each edge represents itself and its reverse in the bidirected graph. If we view a path as a set, as the order has little importance here, with slight abuse of notation the property can be written as

$$(\texttt{to\_N} \circ \texttt{path}) \subseteq (\texttt{to\_bidirect}\ UG)$$

where $path$ is returned by $find\text{-}path \circ to\_C \circ UG$, and $to\_bidirect$ converts UG to its bidirected version. Downward preservation of correctness of $find\text{-}path$ implies that, if a path(edge) is returned by $find\text{-}path$,

$$(\texttt{to\_N} \circ \texttt{path}) \subseteq (\texttt{to\_N} \circ \texttt{to\_C} \circ UG)$$

It appears that we only need a lemma of the following form:

$$\forall\ G, (\texttt{to\_N} \circ \texttt{to\_C} \circ \texttt{G}) \subseteq (\texttt{to\_bidirect}\ G),$$

which is very easy to prove. This lemma says that `to_N` and `to_C` preserve information in the input graph to some extent, and we will discuss more in the following section.

At this point, we conclude the correctness of the path-finding algorithm.

D. *Identity and Correctness of* `to_N` *and* `to_C`

Although we proved the correctness of the algorithm, we still need evidence to support the correctness of `to_N` and `to_C`. An extreme example is that, if we change the implementation of `to_C` to let it always return an empty list, then it is obvious that the correctness properties of the algorithm are vacuously true. This is due to `to_C` losing information about the graph. The strongest property that one desires is completeness, i.e. the algorithm returns all correct paths. But the proof is beyond the scope of this work. Instead, we prove that $\texttt{to\_N} \circ \texttt{to\_C}$ is "almost" an identity map.

We have seen that

$$\forall\ G, (\texttt{to\_N} \circ \texttt{to\_C} \circ \texttt{G}) \subseteq (\texttt{to\_bidirect}\ G)$$

from the previous section, if we view a graph as a set and ignore the ordering of its elements. One would expect the inverse to be true, so that to_N and to_C perfectly preserve information of the input graph:

$$(\texttt{to\_N} \circ \texttt{to\_C} \circ \texttt{G}) = (\texttt{to\_bidirect}\ G)$$

. However that is not the case. The inverse is true only if additional constraints are present:

$\forall e \in G,$
$(G$ has no self loop$)\wedge$
$(\exists p\_e \in (\texttt{to\_bidirect}\ G), p\_e$ is a previous edge of $e)\wedge$
$(e$ is not an input edge$) \implies$
$e \in (\texttt{to\_N} \circ \texttt{to\_C} \circ G).$

This theorem says that an edge in an undirected graph G is also in $(\texttt{to\_N} \circ \texttt{to\_C} \circ G)$ if:

- G has no self loop
- There is a previous edge p_e of e in the bidirected version of G. This proves the existence of an arc corresponding to e in $\texttt{to\_C} \circ G$, hence e is recovered when applying `to_N` to that arc. This condition is necessary, otherwise e is an obsolete path and thus has no previous edge to form an arc with.
- e is not an input edge, i.e., it does not start nor end at the input vertex. This is also necessary, as `to_C` does not generate arcs corresponding to an edge that ends at input vertex; and an edge starting from input vertex does not have a previous edge.

These conditions are very easily met by most of the edges in a real airport. Therefore we conclude that `to_N` and `to_C` preserves information of the input graph to a good extent, and the implementation of these algorithms are desired. However, this proof does not justify the completeness of the entire algorithm, which is beyond the scope of this paper.

V. Discussion

A. *Generality*

The correctness of our algorithm is guaranteed by our proofs in Coq, so the algorithm can be generally applied on arbitrary airports besides KARB (Ann Arbor airport). We use KARB in this paper mainly for introductory purpose, namely showing what we're doing in each step and providing an intuitive understanding of the correctness of our propositions. We also successfully ran our algorithm on bigger airports such as KYIP (Willow Run Airport) and KDTW (Detroit Metropolitan Wayne County Airport, a large international airport).

B. *Limitations*

1) Completeness: We only proved the soundness of our algorithm, i.e. the algorithm only outputs correct paths from the input. We have not proved completeness, i.e. that the algorithm outputs every correct path, and we leave this proof for future work.

2) Recursion Step Bound: One weakness of our implementation is the introduction of a hard-coded bound of maximal recursion steps in the find-path algorithm Algorithm 2. Coq requires every recursive function to be well-founded, for which the bound is the evidence. Having this bound potentially limits the result to exclude long paths. But real airports are relatively small compared to the computation capacity of computers, hence a sufficiently large bound would not cause problems in real usage. We proved that the number of recursion steps is equal to the length of $s@1$, the traveled arcs, so the value of the bound is provably the maximal allowed path length. We set the default value to 100, which is enough for most airports. In real cases, the ATC command won't give instructions to go in circles, so the aircraft goes through every edge at most one time. It means the aircraft will traverse the airport after $|E|$ round of iteration, while $|E|$ means the number of edges in the undirected graph encoding of the airport. Setting the round bound to $|E|$ is quite safe but unnecessary, because the ATC command will definitely not instruct the aircraft to such an extreme path.

One way to get rid of the bound is to provide more sophisticated evidence that the function is well-founded. We give a proof sketch but leave the actual proof to future work. Each step of the find-path algorithm Algorithm 2 either adds a new edge from the current taxiway or from the next taxiway. If one sticks to a single taxiway name, since one cannot take the previous edge as the next edge, the algorithm traverses the same taxiway in the same direction; and if no taxiway forms a loop, the algorithm will eventually reach an end of this taxiway. Therefore, before moving on to the next taxiway, it takes L steps at most, where L is the length of the longest taxiway. Additionally, the ATC command has a finite number of taxiway names, thus the maximum number of recursion steps is also finite.

3) Performance: The algorithm is designed to be proof-friendly, and as discussed before, the input in reality is small, hence performance is not an important factor. Additionally, pruning happens rapidly in reality, since one can only take the current taxiway or move to the next specified taxiway, and the latter is relatively rare. Informally, this makes the time complexity close to $O(L)$ where L is the length of the ATC command, instead of exponential in L for a normal breadth-first search algorithm. Redundant data can also be removed. The purpose of $s@4$ for a state s, which denotes the used ATC commands, is to simplify the proof, but it has little effect on the algorithm's behavior. Writing a new algorithm without these redundancies with a proof of their isomorphism can improve the space complexity.

C. Lessons Learned

We modularized the algorithm and the proof, making them easy to fix and rewrite. The nature of Coq does not give us much freedom to refactor the code, and some legacy problems such as suboptimal data structures and confusing logic remain. Some proofs of some theorems were duplicated during the development process.

Powerful automation such as algorithm specific tactics, setoid rewriting and the hammer tactic [19] made the proof easier and more concise.

Acknowledgments

The authors would like to thank Necmiye Ozay and Yuhao Zhang for their comments. This work is supported in part by Collins Aerospace, a unit of United Technologies Corp. The authors would like to thank Eelco Scholte and Timothy J. Woods from Collins Aerospace for helpful discussions. The article solely reflects the opinions and conclusions of its authors and not Collins Aerospace.

References

[1] S. D. Young and D. R. Jones, "Runway incursion prevention: A technology solution," 2001, nASA Technical Report.

[2] J. Schönefeld and D. Möller, "Runway incursion prevention systems: A review of runway incursion avoidance and alerting system approaches," *Progress in Aerospace Sciences*, vol. 51, pp. 31–49, 2012.

[3] V. H. L. Cheng, V. Sharma, and D. C. Foyle, "A study of aircraft taxi performance for enhancing airport surface traffic control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 2, no. 2, pp. 39–54, 2001.

[4] C. Liu and S. Ferrari, "Vision-guided planning and control for autonomous taxiing via convolutional neural networks," in *AIAA Scitech 2019 Forum*, 2019, p. 0928.

[5] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI," in *International Conference on Computer Aided Verification (CAV)*, 2020.

[6] K. D. Julian, R. Lee, and M. J. Kochenderfer, "Validation of image-based neural network controllers through adaptive stress testing," in *International Conference on Intelligent Transportation Systems (ITSC)*, 2020.

[7] B. Lu, M. Coombes, B. Li, and W. Chen, "Improved situation awareness for autonomous taxiing through self-learning," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 12, pp. 3553–3564, 2016.

[8] W. Eaton and W. Chen, "Image segmentation for automated taxiing of unmanned aircraft," in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2015, pp. 1–8.

[9] C. Muñoz, V. Carreño, G. Dowek, and R. Butler, "Formal verification of conflict detection algorithms," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 3, pp. 371–380, May 2003. [Online]. Available: https://doi.org/10.1007/s10009-002-0084-3

[10] P. Nuzzo, H. Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donzé, and S. A. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2013.

[11] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, "A formally verified hybrid system for the next-generation airborne collision avoidance system," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 21–36.

[12] S. Balachandran, N. Ozay, and E. M. Atkins, "Verification guided refinement of flight safety assessment and management system for takeoff," *Journal of Aerospace Information Systems*, pp. 357–369, 2016.

[13] S. A. da Costa and L. Ribeiro, "Formal verification of graph grammars using mathematical induction," *Electronic Notes in Theoretical Computer Science*, vol. 240, pp. 43 – 60, 2009, proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066109001662

[14] O. Kupferman, "Examining classical graph-theory problems from the viewpoint of formal-verification methods (invited talk)," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, ser. STOC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 6. [Online]. Available: https://doi-org.proxy.lib.umich.edu/10.1145/3055399.3079075

[15] The Coq Development Team, "The Coq Proof Assistant, version 8.10.0," Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3476303

[16] Y. Zhang, G. Poupart-Lafarge, H. Teng, J. Wilhelm, J.-B. Jeannin, N. Ozay, and E. Scholte, "A software architecture for autonomous taxiing of aircraft," in *AIAA Scitech 2020 Forum*, 2020, p. 0139.

[17] Federal Aviation Administration, "Ann Arbor Municipal (KARB) Airport Diagram," https://aeronav.faa.gov/d-tpp/2005/05506AD.PDF, retrieved May 2020.

[18] F. Harary and R. Z. Norman, "Some properties of line digraphs," *Rendiconti del Circolo Matematico di Palermo*, vol. 9, no. 2, pp. 161–168, 1960.

[19] Ł. Czajka and C. Kaliszyk, "Hammer for coq: Automation for dependent type theory," *Journal of automated reasoning*, vol. 61, no. 1-4, pp. 423–453, 2018.