A Fast and Verified Algorithm for Proving Store-and-Forward Networks Deadlock-Free

Abstract—Deadlocks are an important issue in the design of interconnection networks. A successful approach is to restrict the routing function such that it satisfies a necessary and sufficient condition for deadlock-free routing. Typically, such a condition states that some (extended) dependency graph must be acyclic. Defining and proving such a condition is complex. Proving that a routing function satisfies a condition can be complex as well. In this paper we present the first algorithm that automatically proves routing functions deadlock-free for store-and-forward networks. The time complexity of our algorithm is linear in the size of the resource dependency graph. The algorithm checks a variation of Duato's condition for adaptive routing. The condition and the algorithm have been formalized in the logic of the ACL2 interactive theorem prover. The correctness of our algorithm w.r.t. the condition is formally checked using ACL2.

I. Introduction

Deadlocks occur in interconnection networks when packets wait for channels or buffers that will never be available. This is an important aspect in the design of protocols and architectures, in particular in the definition of the routing function. The study of necessary and sufficient conditions for deadlock-free routing has been a very active and successful research area [1], [2], [3], [4], [5]. The resource (channel or buffer) dependency graph captures the dependencies introduced by the routing function. In the case of deterministic functions, an acyclic dependency graph is necessary and sufficient to prevent the creation of deadlocks. In the case of adaptive functions, Duato [6], [3] showed that routing functions with cyclic dependencies can still be deadlock-free. His condition states that the existence of a routing subfunction with an acyclic extended dependency graph is both necessary and sufficient.

Until recently there was no algorithm that would automatically decide if a routing function satisfies a necessary and sufficient condition. Systematic methods were proposed (e.g. [6]) but they are based on a manual process and are therefore error-prone. In the case of complex routing functions, discharging the condition is a difficult process as well.

To the best of our knowledge, Taktak et al. [7], [8] have been the first to propose a practical algorithm to prove networks deadlock-free. Their work applies to wormhole networks. The authors define their own necessary and sufficient condition and provide an algorithm to automatically check this condition. Let $N$ be the number of processing nodes of a network. In the worst case, there may be dependencies between any pair of nodes. The size of the resource dependency graph is therefore in $O(|N|^2)$. The time complexity of Taktak's algorithm is $O(|N|^4)$, i.e., quadratic in the size of the dependency graph. The algorithm is proven correct w.r.t. the condition by a pencil and paper proof.

Necessary and sufficient conditions for deadlock-free routing are complex and often counterintuitive. Taktak's condition and algorithm are no exceptions. A complex mathematical proof is required to justify a condition or an algorithm. This pencil and paper proof is a manual process and therefore error-prone. In [9] we show that Taktak's condition is only sufficient and that their algorithm is subject to false negatives, i.e., identification of unreachable deadlock situations.

Since the mid 70's, interactive theorem provers have been designed to mechanically check formal and detailed proofs. Their development and application in various domains are active research fields. These proof assistants are used in projects about formalizing mathematics (e.g. the FlySpeck project [10]) or in the verification of hardware and software designs (e.g. microprocessors [11], [12], [13], floating point units [14], [15], [16], [17], on-chip networks [18], operating systems [19], entire computing systems [20]). The most popular tools are ACL2 [21], Isabelle [22], PVS [23], Coq [24], HOL [25], HOL-Light [26].

Our utmost objective is to implement an efficient algorithm that would be formally proven to check a formally verified necessary and sufficient condition for deadlock-free routing. The use of a mechanical proof assistant increases the confidence in a condition and the correctness of its associated algorithm. To reach this goal one has to (1) formally define a deadlock configuration for a class of networks, a condition for deadlock-free routing, and prove that the condition is necessary and sufficient to build a deadlock configuration; (2) formally define an algorithm and prove that this algorithm identifies a deadlock if and only if the condition holds.

In the context of store-and-forward networks, we recently used the ACL2 theorem prover to define and formally check a necessary and sufficient condition for adaptive [27] routing functions. The contribution of this new paper is a novel algorithm checking the latter condition with a time complexity that is linear in the size of the resource dependency graph. The correctness of the algorithm w.r.t. to our formally verified condition has been formally verified using ACL2.

The rest of the paper is organized as follows. In the next section we briefly define our necessary and suffi-
cient condition for adaptive routing functions in store-
and-forward networks. This section also formally defines
our network model, the dependency graph, and a dead-
lock configuration. Section III introduces our algorithm
in two steps. We first run the algorithm on a simple
example before giving its formal definition. We sketch
the proof of the time complexity and the correctness of
the algorithm in Section IV. In this Section, we briefly
discuss the mechanical proof in ACL2. We relate our
work to previous results in Section V. Finally, Section VI
discusses the mechanical proof in ACL2. We relate our
work to previous results in Section V. Finally, Section VI
presents our conclusions and future research directions.

II. A necessary and sufficient condition

An interconnection network consists of processing
nodes connected by channels. These nodes consist of
ports and a central switch (see Figure 1). The switch
contains the routing function and the switching method.
There is a port for each in- and outgoing channel.
Each node has a local in- and out-port, respectively for
injecting and removing messages from the network. With
each port a list - of size at least 1 - of buffers is associated.
One buffer can store one packet. Bufferless switching
can be represented by associating exactly one buffer per
port. Furthermore, we assume that all destination ports
(i.e., local output ports) are terminal, i.e., they are not
connected to other ports. A destination port is therefore
never blocked.

A. Formal model

1) Interconnection network: An interconnection net-
work is a directed graph, \( I = (P, E) \). The vertices \( P \)
of the graph represents the set of ports. The arcs \( E \) repre-
sent the topological connections between the ports. Each port
\( p \) has a certain buffer capacity to store messages.

2) Routing and dependency graph: An adaptive routing
function \( R : P \times P \mapsto \mathcal{P}(P) \) takes as parameters the port
a message currently occupies and the destination of the
message. It returns a set of ports the message can use to
get from the current port to its destination, i.e. the next
hops. A destination \( d \) is reachable from port \( s \), notation
\( s \to d \), if and only if the routing function can supply a
path from \( s \) to \( d \) for messages destined for \( d \).

The port dependency graph depicts the dependencies
between the ports of the interconnection network. It is a
diagram, where each port has two buffers.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{processing_node}
\caption{Processing node, where each port has two buffers.}
\end{figure}

graph \( \text{dep} = (P, E_{\text{dep}}) \). The vertices are the set of ports \( P \).
The arcs \( E_{\text{dep}} \) are the pairs of ports \((p_0, p_1)\) connected by
the routing function.

Given function \( E_{\text{dep}} \), we define the overloaded function
\( E_{\text{dep}} : P \times P \mapsto \mathcal{P}(P) \) as follows:

\[ E_{\text{dep}}(p, d) = \{ n \in E_{\text{dep}}(p) \mid n \in R(p, d) \} \]

When given extra parameter \( d \), overloaded function \( E_{\text{dep}} \)
returns a subset of the set of neighbors. \( E_{\text{dep}}(p, d) \) returns
the set of dependency neighbors created by destination
\( d \), i.e., the set of next hops for a message located in \( p \)
and destined for \( d \).

3) Switching method: We have formalized store-and-
forward Packet Switching. A packet contains data and a
header. The header stores information on the destination
of the packet. We denote the destination of packet \( m \) with
dest(\( m \)). At each port the packet is stored, analyzed, and
if possible forwarded to other ports, i.e. the next hops.

Packets are the atomic objects transferred between any
two ports. We assume that the number of packets a port
can hold is arbitrary but fixed. Function \( \text{cap} : P \mapsto \mathbb{N}^+ \)
returns the number of packets a port can store. A packet
is blocked if and only if all its next hops are unavailable. A
port \( p \) is unavailable if and only if it stores \( \text{cap}(p) \) packets.
If multiple next hops are available, one of them is
chosen arbitrarily. For an in-depth discussion on packet
switching we refer to a survey by Ni and McKinley [28].

4) Deadlock: A legal configuration \( \sigma \) is an assignment of
packets to ports, where the capacity of each port is not
exceeded. Given a port \( p \), \( \sigma(p) \) returns the set of packets
stored in \( p \) and \( |\sigma(p)| \) denotes the number of packets
stored in \( p \). Formally, \( \sigma \) is a deadlock configuration if
and only if it is a non-empty configuration such that:

\[ \forall p \in P \cdot \{ |\sigma(p)| \leq \text{cap}(p) \} \]

A deadlock configuration is a legal configuration where
all packets are blocked, i.e., where for all packets all next
hops are unavailable.

5) Assumptions: Our condition is developed under the
following assumptions:

1) Messages are destined for reachable destinations
   only.
2) Destinations are sinks, i.e., they do not store mes-
   sages that need to be forwarded to other ports.
3) Packets that have arrived at their destination will
   eventually be consumed.

B. A necessary and sufficient condition

The condition used in this paper is based on the notion
of escapes. Given a set of ports \( P' \) a port \( e \in P' \) is an
escape for \( P' \) if and only if for all possible destinations
there exists a dependency neighbor that is not contained
in \( P' \):

\[ \text{esc}(e, P') \overset{\text{def}}{=} \forall d : E_{\text{dep}}(e, d) \not\subseteq P' \]
In other words, an escape for a subgraph is a port in which any message can be routed outside of the subgraph regardless of the destination.

Consider the set of black ports in Figure 2(a). This subgraph has no escape: port A has no $d_1$-edge outside of the subgraph and likewise for port C and destination $d_0$. Indeed, if (a) ports $A$, $B$, $C$ are full, (b) all messages in $A$ are destined for $d_1$, and (c) all messages in $C$ are destined for $d_0$, then the result is a deadlock-configuration. If we add $d_0$ to edge $(C, D)$ such as in Figure 2(b) port C becomes an escape. For both destinations $d_0$ and $d_1$ there exists a neighbor (namely $D$) outside the subgraph. There is no deadlock-configuration possible in this network.

Let $P$ be some set of ports in deadlock. Subgraph $P$ cannot contain an escape port, since otherwise at least one packet in this escape port could move to a port not in $P$. The condition used in this paper formulates this contrapositively:

Theorem 1: An interconnection network is deadlock-free if and only if all subgraphs of the port dependency graph have an escape.

Theorem 1 has been formally proven correct [27]. In the next section we define an algorithm checking Theorem 1 in linear time.

III. A linear algorithm

A. The algorithm by example

The basic objective of the algorithm is to mark each port as deadlock-immune or deadlock-sensitive. After termination of the algorithm, it is possible to create a deadlock by filling all deadlock-sensitive ports. In this case the algorithm outputs the exact reason for the deadlock. If all ports are marked as deadlock-immune, the network is deadlock-free.

We first demonstrate by example the notions of deadlock-immunity and -sensitivity. Secondly we provide an example trace of the algorithm, showing how it detects a deadlock. Lastly, we provide an example trace that reveals the necessity of a post-processing step.

1) Deadlock-immunity and -sensitivity: Consider the interconnection network in Figure 3(a). Nodes $d_0$ and $d_1$ are the only possible destinations. An edge $(p_0, p_1)$ is labeled $d$ if destination $d$ leads from $p_0$ to $p_1$. An unlabeled edge means that any destination leads from $p_0$ to $p_1$. Figure 3(b) shows the result of our algorithm. A marking $x | y$ stands respectively for the deadlock-sensitive and the deadlock-immune destinations of that port. The white (black) ports are deadlock-immune (sensitive). The definitions of deadlock-immunity and -sensitivity are cyclic. Therefore they are not formally defined, They are only used for a better understanding of the algorithm.

Nodes $d_0$ and $d_1$ are sinks and by definition deadlock-immune, i.e., they can never be used to form a deadlock. At port $D$ messages can either reach sinks or be routed to port $A$. The former provides for all possible destinations escapes to a potential deadlock. Our algorithm therefore marks port $D$ as deadlock-immune. Ports $A$, $B$ and $C$ are marked as deadlock-sensitive as a deadlock can be constructed by filling these ports with packets. Port $B$ is filled with messages destined for $d_1$, port $C$ is filled with messages destined for $d_0$, and port $A$ is filled with any message.

We now detail how this result is produced by our algorithm.

2) Example trace: The algorithm consists of two steps. The first step expands a spanning tree. After expanding the tree forwards, information is propagated backwards.
The second step performs some post-processing.

Consider the network in Figure 3(a). Let the algorithm start in port A. The different steps of the run of the algorithm are shown in Figure 4. The algorithm expands a tree spanning over the reach of A. It starts by marking port A as visited (Step 1). All destinations lead to B. Thus, the next step is to expand B, i.e., mark it as visited (Step 2). To determine deadlock-immunity of port B, both ports C and D must be considered.

Consider the expansion of C (Step 3). Destination d1 is deadlock-immune for port C as it leads to a sink. Destination d0 leads back to a visited port. In this case it is considered deadlock-sensitive for port C. A port is only deadlock-immune if for all deadlock-sensitive destinations there exists an escape, i.e., a deadlock-immune counterpart. Port C is a bottom leaf of the spanning tree, since all neighbors are either sinks or visited. Since we have arrived at a leaf, information is propagated backwards to port B: all destinations leading to C are deadlock-sensitive for port B (Step 4).

Consider the expansion of D (Step 5). Both destinations are deadlock-sensitive for port D, since both destinations lead back to a visited port. However, both destinations are deadlock-immune for port D as well, since both lead to a sink. Since each deadlock-sensitive destination has an escape, i.e., a deadlock-immune counterpart, port D is deadlock-immune. Since port D is a bottom leaf, information is again propagated backwards to port B: all destinations leading to D are deadlock-immune for port B (Step 6).

At this point all neighbors of port B have been expanded. All information on them has been propagated backwards: destination d1 is deadlock-sensitive and destination d0 is deadlock-immune. There is a deadlock-sensitive destination d1 for port B that is not deadlock-immune, thus port B is deadlock-sensitive. Again information is propagated backwards to port A: all destinations leading to B are deadlock-sensitive for port A (Step 7). For port A all destinations are deadlock-sensitive and they have no deadlock-immune counterpart. It is marked as deadlock-sensitive.

The algorithm terminates and there are ports marked as deadlock-sensitive. A deadlock can be created by filling all deadlock-sensitive ports with messages destined for destinations that are deadlock-sensitive and do not have a deadlock-immune counterpart.

3) Post-processing: Consider the network in Figure 5(a). Note that this network is deadlock-free. It has only one cycle and this cycle has an escape, namely port A. For all reachable destinations, a message in port A can escape the cycle.

Assume the first step of the algorithm starts in port A and marks it as visited. In this specific trace, the first neighbor of A that is expanded is port B. Destination d1 is deadlock-immune for port B as it leads to a sink. Destination d0 is deadlock-sensitive for port B as it leads back to visited port A. Port B gets marked as deadlock-sensitive, since there is a deadlock-sensitive destination that is not deadlock-immune. As a result, destination d1 is deadlock-sensitive for port A. The algorithm continues with the expansion of the other neighbors of A. Since both destinations d0 and d1 lead to a sink, they are marked as deadlock-immune for port A. The deadlock-sensitive destination d1 is included in the set of deadlock-immune destinations \{d0, d1\}. Thus port A is marked deadlock-immune. Port B however, is still marked as deadlock-sensitive.

If the algorithm would stop here, it would conclude that this network is not deadlock-free as not all ports have been marked as deadlock-immune. The problem is that in this specific trace at the time d0 was marked deadlock-sensitive for port B, port A had not been expanded completely. In other words, it was not known at the time that port A is deadlock-immune. To overcome this, we add a post-processing step to the algorithm. This step adds for all deadlock-sensitive ports all destinations leading to deadlock-immune ports. In this step, a deadlock-sensitive port can become deadlock-immune. See Figure 5(c).

After the post-processing step, a deadlock can be formed of all ports that are still deadlock-sensitive. A network is deadlock-free if and only if after termination of the two steps of the algorithm all ports are marked deadlock-immune.

B. Formal description

We assume a dependency graph with each edge \((p_0, p_1)\) labeled with the destinations that lead from \(p_0\) to \(p_1\). Let function \(\Delta(p_0, p_1)\) return the labels of edge \((p_0, p_1)\).

Let us consider step 1 (Algorithm 1), named CreateTree. Let \(P_1\) be a set of unmarked ports. The algorithm keeps expanding new neighbors until no unmarked neighbors exist. The current port under investigation is \(p_0\). The algorithm keeps track of the parent of \(p_0\) in parameter \(f\). This enables the backwards propagation. For each port, the algorithm stores the deadlock-immune destinations in array imm and the deadlock-sensitive
destinations in array sens. A port can get four possible markings:

0 A port is unmarked;
1 A port is under investigation, i.e., not all neighbors have been marked;
2 All neighbors have been marked, the port is deadlock-immune;
3 All neighbors have been marked, the port is deadlock-sensitive.

If \( p_0 \) is either a sink or deadlock-immune (line 6), then all destinations \( \Delta(f, p_0) \) are deadlock-immune for \( f \). Thus these destinations are added to \( \text{imm}(f) \). If \( p_0 \) is marked otherwise (line 3) two cases arise: either \( p_0 \) has already been shown to be deadlock-sensitive or it is unknown at this point whether \( p_0 \) is deadlock-sensitive or immune. In both cases, all destinations \( \Delta(f, p_0) \) are considered deadlock-sensitive for port \( f \) and these are added to \( \text{sens}(f) \).

If \( p_0 \) is neither a sink or marked, the algorithm continues its forwards expansion by expanding the neighbors of \( p_0 \) (line 12). When this terminates, the gathered information is propagated backwards through the graph as follows: if there exists a destination in \( \text{sens}(p_0) \) that is not in \( \text{imm}(p_0) \), port \( p_0 \) is deadlock-sensitive. Thus \( \Delta(f, p_0) \) is added to \( \text{sens}(f) \) and \( p_0 \) is marked with 3 (lines 17-18). If all destinations in \( \text{sens}(p_0) \) are included in \( \text{imm}(p_0) \), port \( p_0 \) is deadlock-immune and \( \Delta(f, p_0) \) is added to \( \text{imm}(f) \). Port \( p_0 \) is marked with 2 (lines 14-15).

As for the post-processing step (Algorithm 2), this step initially considers all 3-marked ports with 2-marked neighbors. For all these ports it adds the destinations leading to 2-marked neighbors, which have not been added already (line 4). If, as a result of this, a port \( p \) gets marked 2 (lines 5-6), then all parents of \( p \) have a new 2-marked neighbor. Thus, all parents must be reconsidered (line 7).

Main wraps up the two steps. It executes CreateTree for all unmarked ports \( p \), with \( P_I = \{E_{\text{dep}}(p)\} \) and \( f = p \). After this, it executes Post-Processing.

### IV. Analysis

**A. Running time**

CreateTree visits each unmarked port exactly once, since after visitation a port becomes permanently marked. The total running time of all calls of this step is therefore \( O(|P|) \). It is basically a depth-first search, with backwards propagation.

The running time of the post-processing step is \( O(|E|) \) with \( |E| \) the number of edges in the dependency graph. Let the algorithm start with 32-edges only, i.e., all edges starting in a 3-marked port \( s \) and ending in a 2-marked port \( d \). For all 32-edges, the algorithm adds the labels to the imm-array of the source \( s \) (line 4). The edges considered in line 4 are considered once: they could be permanently removed from the data-structure storing the graph. Line 7 adds new parents to \( P_I \), thereby adding new edges that are to be taken into consideration.

All these edges were initially 33-edges, but have just become 32-edges as port \( p_0 \) has just been marked 2. As the algorithm only considers 32-edges, none of these new edges have been dealt with before. Each edge is considered at most once.

The running time of Main is the sum of the running times of CreateTree and Post-Processing. Before post-processing, Main filters all 32-edges. This can be done in \( O(|E|) \). After post-processing it searches for a 3-marked port in \( O(|P|) \). The total running time of the algorithm is \( O(|E|) \).

**B. Correctness**

We prove that our algorithm returns true if and only if our condition holds.

1) **Proof sketch:** The proof is structured in two parts: Lemma 3 states that our algorithm is sufficient for
Algorithm 3 Main

Require: $P = \{p_0, p_1 \ldots p_k\}$, with $k > 0$
1: for $i = 0$ to $k$ do
2: if $\text{marks}(p_i) = 0$ then
3: $\text{CreateTree}(E_{\text{dep}}(p_i), p_i)$
4: end if
5: end for
6: $\text{Post-Processing}(\{p \in P \mid \text{marks}(p) = 3 \land \exists p' \in E_{\text{dep}}(p) \cdot \\
\text{marks}(p') = 2\})$
7: return $\{p \in P \mid \text{marks}(p) = 3\} = \emptyset$

Lemma 2: After termination of Post-Processing, if for any port $p$ all reachable destinations lead to a 2-marked neighbor, port $p$ is not marked 3.

Proof: The post-processing step ensures that for all 3-marked ports the imm array contains all destinations leading to 2-marked neighbors. Since, by assumption, all destinations reachable from $p$ lead to a 2-marked neighbor, all reachable destinations are included in imm($p$). Thus necessarily sens($p$) $\subseteq$ imm($p$), which implies the port can never become marked 3.

We now prove sufficiency: in a deadlock-free network, any port gets marked 2. Thus, the algorithm will return true if the condition for deadlock-freedom holds.

Lemma 3: Assume all non-empty subgraphs have an escape. After termination of Post-Processing() any port $p$ is marked 2.

Proof: By Lemma 1, port $p$ is either marked 2 or 3. The proof is by contradiction. Assume port $p$ is marked 3. We prove that the set of 3-marked ports does not have an escape. Let $p'$ be any 3-marked port. By Lemma 2 there is at least one destination $d$ that does not lead to any 2-marked neighbor. By Lemma 1 destination $d$ leads to 3-marked ports only. Since there is a destination that does not lead outside of the subgraph consisting of all 3-marked ports, port $p'$ is not an escape for this subgraph. This holds for all $p'$ in the subgraph. Thus the subgraph does not have an escape. Furthermore this subgraph is not empty, since otherwise there would be no 3-marked ports and port $p$ is marked 3. Thus the assumption that all non-empty subgraphs have an escape has been contradicted.

Lastly, the part of necessity is proven. If the algorithm returns true, i.e., if all ports get marked 2, then the condition for deadlock-free routing holds. In other words, if the condition does not hold, there is some port that will not be marked 2.

Lemma 4: If a port $p$ is in a subgraph $S$ that has no escape, the port will not be marked 2.

Proof: The lemma holds initially since all ports are unmarked. We show by induction on CreateTree that this lemma is preserved during this step. The exact similar argument holds for Post-Processing. Thus the lemma is an invariant for the algorithm.

Assume that port $p \in S$ and that $S$ has no escape. The only reason a port $p$ gets marked 2 is when sens($p$) $\subseteq$ imm($p$). We prove that this implies port $p$ is an escape. When port $p$ becomes marked 2 all neighbors of port $p$ have been expanded. Thus all reachable destinations are either in sens($p$) or in imm($p$). Since by assumption sens($p$) $\subseteq$ imm($p$), all reachable destinations are in imm($p$). If a destination is in imm($p$) then it leads to a 2-marked neighbor. Thus for all reachable destinations, there is a 2-marked neighbor. By the Induction Hypothesis, none of the ports in subgraph $S$ are marked 2. Thus for all reachable destinations there is a neighbor not in the subgraph: $p$ is an escape for the subgraph. This contradicts the assumption that $p$ is in a subgraph without an escape. Thus port $p$ cannot have been marked.

deadlock-freedom and Lemma 4 states that it is necessary. The proofs of these lemmas require Lemmas 1 and 2.

Lemma 3 is proven as follows. If the algorithm marks a port 3 and this marking is preserved by the post-processing step, it is possible to create a subgraph without an escape. This proof completely formalizes the intuition in Figure 3(a): a deadlock is created from all deadlock-sensitive ports. Subgraph $S_3$ is created by taking all 3-marked ports. Each port $p$ in subgraph $S_3$ has a destination $d$ that is deadlock-sensitive, but not deadlock-immune. Since deadlock-sensitive (immune) destinations lead to 3-marked (2-marked) neighbors, destination $d$ only leads to 3-marked ports (Lemma 2). Since subgraph $S_2$ contains all 3-marked ports and since port $p$ has destination $d$ which leads to 3-marked ports only, port $p$ is not an escape for this subgraph. Since this holds for all ports $p$ in subgraph $S_2$, the subgraph has no escape.

Lemma 4 states that any port in a deadlock cannot be marked 2. Since all ports eventually are marked either 2 or 3 (Lemma 1), any port that can be in a deadlock gets marked 3.

Together, Lemmas 3 and 4 state that a port gets marked 3 if and only if it is in some subgraph without an escape. The algorithm returns true if and only if all ports get marked 2. By Theorem 1, the algorithm returns true if and only if the network is deadlock-free.

2) Proof: The lemmas of the proof concern 2- and 3-marked ports. First we prove that any port eventually gets one of these markings.

Lemma 1: After termination of CreateTree($E_{\text{dep}}(p_0), p_0$) all ports in the reach of the ports in $P$ are either marked 2 or 3.

Proof: Any unmarked port in the reach will eventually get marked 1. Any 1-marked port will eventually become marked either 2 or 3. A port is only marked 1 on line 11. Eventually the algorithm will reach either line 15 or line 18, where the port is marked either 2 or 3. Once a port is marked 2 or 3, it will never become either unmarked or marked 1.

We prove that if a port has escapes for all reachable destinations, i.e., if all reachable destinations lead to a deadlock-immune neighbor, the port will not be marked 3. This lemma requires the post-processing step.
These lemmas suffice to prove that the algorithm decides the necessary and sufficient condition of Section II-B.

**Corollary 1:** A network is deadlock-free if and only if \( \text{Main}(P) \) returns true.

**Proof:** Theorem 1 states that an interconnection network is deadlock-free if and only if all subgraphs have an escape. Assume all subgraphs have an escape. By Lemma 3 all ports will be marked 2 and thus \( \text{Main} \) returns true. Assume \( \text{Main} \) returns true. Then all ports are marked 2. By Lemma 4 there is not a port in a subgraph without an escape. Thus all subgraphs have an escape.

### C. ACL2 formalization

ACL2 [21] stands for "A Computational Logic for Applicative Common Lisp". It denotes a logic, a programming language, and a mechanized theorem prover. The logic is a first order logic with induction and a definitional principle allowing users to safely extend the logic with new function symbols and axioms. The programming language is a side-effect free subset of Common Lisp. ACL2 functions are therefore executable. The theorem prover engine is based on heuristic decisions that can be influenced by users. The interaction with the tool happens by adding lemmas to the logic. These lemmas generate rules of particular types. These rules are then used by the theorem prover in further proofs. We chose ACL2 because its logic is expressive enough for our purpose and it provides a fairly high level of automation. In contrast, many other systems have a more expressive logic with a lower degree of automation.

Table 1 gives details on the total effort of implementing and proving correctness of the algorithm. The first two lines concern the contribution of this paper. The other lines concern preliminary work that was needed for this proof [18], [27].

The file containing the algorithm also contains formal verification of all guards. This includes proofs of termination of the algorithms and proofs that, if a valid dependency graph is supplied, the algorithm will always properly access and write to its data structures. Guard verification is needed to make the functions executable in Common Lisp.

Table 1 Details on the verification effort

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
<th>Theorems</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm/Guard verification</td>
<td>1405</td>
<td>157</td>
<td>27</td>
</tr>
<tr>
<td>Proof of correctness</td>
<td>2865</td>
<td>233</td>
<td>31</td>
</tr>
<tr>
<td>Proof of condition</td>
<td>2540</td>
<td>243</td>
<td>47</td>
</tr>
<tr>
<td>Dependency graph</td>
<td>423</td>
<td>23</td>
<td>37</td>
</tr>
<tr>
<td>Generic Routing Function</td>
<td>202</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>Packet Switching</td>
<td>1022</td>
<td>107</td>
<td>22</td>
</tr>
</tbody>
</table>

The second file contains a proof of correctness, i.e. a formal proof that the implementation of our algorithm returns true if and only if Theorem 1 holds.

### V. Related work

To the best of our knowledge, the only other polynomial algorithm deciding deadlock-freedom for adaptive routing functions is recently created by Taktak [7].

1) Taktak’s algorithm applies to wormhole networks whereas our algorithm applies to store-and-forward networks. A deadlock in a wormhole network is not necessarily a deadlock in a store-and-forward network, and vice versa. These two cases require different conditions and therefore different algorithms. Taktak’s algorithm cannot be easily modified to handle store-and-forward networks. Our algorithm checks a necessary and sufficient condition whereas Taktak’s algorithm checks a sufficient condition only. Deciding a sufficient and necessary condition for wormhole network is NP-complete [9]. Defining an algorithm solving this problem is still an open question.

2) Our algorithm is faster. It is linear in the size of the dependency graph, whereas Taktak’s running time is \( O(|C| \cdot N^2) \) with \( N \) the number of routers (nodes) in the network and \( |C| \) the number of strongly connected components in the dependency graph.

3) Taktak et al. assume that a message cannot make a loop in the network, thereby restricting the use of their algorithm to a subset of the adaptive routing algorithms. We do not have this assumption. It is possible to create deadlock-free adaptive networks where messages can make loops. Silla et al. created a generic design methodology for such networks [29]. These networks consist of a deadlock-free part and a fully adaptive part where messages can freely move. Once a message reserves a deadlock-free channel, it cannot leave the deadlock-free part and will eventually arrive at its destination. Such networks can be made livelock-free by adding a counter to the header of each packet, thereby restricting the time messages can spent in the adaptive part of the network. Our algorithm supports the verification of such networks.

4) Both our algorithm and the condition it checks are formally verified. We could have easily made mistakes without formal verification. For instance, the post-processing step is only needed because it is possible that exactly one specific trace of the first step leads to an incorrect behavior. All other traces mark the ports correctly. This makes testing and debugging the algorithm hard. Formal verification helped us to define the post-processing step correctly.

Schwiebert and Jayasimha define another approach for deciding deadlock-freedom of adaptive routing functions [30]. Their algorithm relies on the waiting graph
which depicts the waiting relation between the channels. Although the waiting graph is static, the dependencies that arise between channels are not. Two edges in the waiting graph may share a common channel, in which case the two dependencies cannot occur simultaneously. Thus the waiting graph has True Cycles and False Resources Cycles. The existence of a True Cycle is a necessary and sufficient condition for deadlock-freedom. Schwiebert and Jayasimha define an algorithm distinguishing True Cycles from False Resource Cycles. This algorithm is however exponential in the worst-case.

VI. Conclusion

The main contribution of this paper is an algorithm that proves a routing function deadlock-free in linear time. It checks a necessary and sufficient condition for adaptive routing functions in store-and-forward networks. The algorithm and the corresponding conditions have been formalized in the logic of the ACL2 theorem proving system. Within this system we produced a formal proof of the theorem stating that the algorithm outputs "no deadlock" if and only if the condition for deadlock-free routing holds.

Our algorithm applies to store-and-forward networks. With a slight modification it should apply to virtual-cut through as well. We are currently extending the current algorithm to check a sufficient condition for wormhole networks in linear time.

We did not focus on efficiency but on the correctness proof of the algorithm. The purpose of the definition of our algorithm in ACL2 is to ensure its correctness. In its current state it is not efficiently executable. But the ACL2 language provides many features to support the development of efficient implementations. Our current work aims at using these features to obtain a formally verified implementation that could actually be applied to realistic examples.

References