Abstract—We present our experience and case studies for proving liveness of communication fabrics. A methodology is developed that reveals ranking structure underlying the state spaces of such fabrics. This enables an efficient proof of liveness using the \textit{k-Liveness} algorithm leveraging this ranking structure. An open-source infrastructure for the \textit{k-Liveness} algorithm has been implemented that has a provision for specifying and leveraging ranking structures in the form of stabilizing constraints. A significant speed-up was achieved with this modified \textit{k-Liveness} framework when proving response property of communication fabrics.

Keywords: formal verification, liveness, communication fabrics

I. INTRODUCTION

Consider the xMAS model shown in Figure 1. xMAS (Executable Micro-Architecture Specification)\cite{1} is a formalism proposed for designing micro-architectural communication fabrics. It consists of a library of structural components each having a well-defined logical semantics. Communication fabrics are constructed by appropriately stitching various xMAS components together. The model of Figure 1 is an example of such a fabric called a virtual channel, henceforth referred to as \textit{VC}. A virtual channel is a standard micro-architectural idiom used extensively in practice, particularly in wormhole switching systems. At the architecture level, \textit{VC} is a network of FIFO buffers that connects \textit{flit sources} \textit{A1} and \textit{A2} to \textit{flit sinks} \textit{sink1} and \textit{sink3} respectively\footnote{A flit (flow-control unit) is a unit of data transfer in communication fabrics, see \cite{2} for details.}. The overall objective of \textit{VC} is to enforce flow control in flit transmission from sources to sinks. When source \textit{A1} wants to send a flit to \textit{sink1}, it makes a request on channel \textit{a1}. If buffer \textit{B1} has a token, this request is forwarded to channel \textit{d1}. Based on various conditions like whether channel \textit{d1} is also making a concurrent request or whether buffer \textit{B3} has an empty slot, the arbiter decides when to issue a grant against the request on channel \textit{d2}. This grant is transmitted to source \textit{A1} which allows \textit{A1} to complete the flit transmission. Similar steps take place when \textit{A2} wants to send a flit to \textit{sink3} and makes a request on channel \textit{a2}. A detailed description of the xMAS formalism is provided in the appendix. This will help the reader to understand at the logic level how different xMAS components in \textit{VC} work together to achieve this functionality. For more details on xMAS primitives, see \cite{1}, \cite{3}, \cite{4}. For further discussion on virtual channels, see \cite{2}, \cite{5}.

In brief, an xMAS fabric is a high-level description of a network of FIFO buffers which are connected together using synchronization logic. Appropriate compilers \cite{1} are used to translate these descriptions into bit-level netlists (i.e. synchronous digital circuits working under a single global clock). For example, \textit{VC} of Figure 1 is translated into a bit-level netlist before performing any analysis proposed in this paper.

Suppose we want to prove that in \textit{VC} “whenever a request for flit transmission is asserted by source \textit{A1} on channel \textit{a1}, it is granted eventually (under fairness assumptions sink\_fair and source\_fair defined below)”. Formally, this property is described by the following linear temporal logic (LTL) formula \(\phi(a_1)\) defined on \textit{VC}:

\[
\phi(a_1) := ((\text{sink\_fair} \land \text{source\_fair}) \Rightarrow \text{response}(a_1))
\]

where

\[
\text{sink\_fair} := \text{GF}(m_1.gnt) \land \text{GF}(m_2.gnt) \land \text{GF}(j_1.gnt) \land \text{GF}(j_2.gnt)
\]

\[
\text{source\_fair} := \text{GF}(i_1.req) \land \text{GF}(i_2.req), \text{ and}
\]

\[
\text{response}(a_1) := \text{G}(a_1.req \Rightarrow \text{F}(a_1.gnt)).
\]

\(\text{G}\) and \(\text{F}\) are standard LTL operators representing temporal modalities ‘always’ and ‘eventually’ respectively. The formulation of \(\phi(a_1)\) uses the following xMAS convention: any directed arrow in Figure 1 represents a channel \footnote{The term virtual channel should not be confused with the xMAS component channel. While the latter is present in any xMAS fabric as connectors, the former means a particular micro-architectural model as shown in Figure 1.}. A channel \(x\) consists of a request signal \(x.req\), a grant signal \(x.gnt\) and a data signal \(x.data\). Signals \(x.req\) and \(x.data\) flow toward the direction of the arrow representing channel \(x\), and signal \(x.gnt\) flows in the reverse direction. \(\phi(a_1)\) is an example of a response property \cite{6} which is a type of liveness property. A similar response property \(\phi(a_2)\) can be defined on \textit{VC} for source \textit{A2}. The work presented in this paper aims at proving response properties of the form \(\phi(\cdot)\) efficiently and scalably on communication fabrics of practical importance, using a bit-level verification engine under (strong) fairness assumptions on appropriate signals.

Since \(\phi(\cdot)\) is a liveness property, we could try to prove it with off-the-shelf bit-level liveness model checkers \cite{7} \cite{8} \cite{9}. Unfortunately, on industrially relevant communication fabrics, these suffer from scalability issues\cite{10}. However, we
found that the underlying state spaces of such communication fabrics have interesting ranking structures. We demonstrate this by analyzing a set of fabrics, and show that the runtime of response verification for these fabrics can be improved significantly if we leverage their ranking structures.

This paper makes the following contributions:

1) The operations of an xMAS communication fabric are shown to give rise naturally to a well-founded ordering in its state space.

2) An algorithm is developed to mine disjunctive stabilizing assertions automatically from the bit-level netlist. These capture ranking structures and are shown to speed up liveness verification.

3) To leverage these ranking structures, an extension is given to the basic $k$-LIVENESS [9] algorithm. Preliminary experiments with this extension show promising results.

4) An implementation of this extended $k$-LIVENESS algorithm as well as our benchmarks as Verilog models and as bit-level netlists (see Section V for details) are available from [11].

The case studies of this paper demonstrate how certain novel invariants (ranking structures) can be extracted from xMAS fabrics to speed up liveness verification. To the best of our knowledge, these fabric invariants have not been studied or reported in the literature before. In addition to speeding up verification, this type of ranking structures may be of independent interest. For example, in our case studies, we focused mainly on credit-based flow-control systems. The logical pattern associated with their ranking structures suggests that they may be inferred also for other communication fabrics and general hardware designs. However, this is left for future study. In addition, our tool can be used to prove general liveness properties while our simple extension enhances the power of the basic $k$-LIVENESS algorithm.

It is interesting to note that this work connects a designer's high-level intent to a low-level verification algorithm. Mining high-level design insights to speed up verification is an important goal in formal verification research. Our work offers interesting case studies by mining ranking structures as high-level design invariants and utilizing them in a low-level liveness verification algorithm. Similar work has been done for low-level safety verification algorithms (see [3] for an example application), but little work has been reported for liveness (see Section VII for further discussion).

The paper is organized as follows. Section II presents the general intuition for ranking structures in finite state systems. Section III provides a motivating example that demonstrates how to discover ranking structures in communication fabrics. Section IV presents how the $k$-LIVENESS algorithm is adapted for verification of response properties and how ranking structures can be used to speed up $k$-LIVENESS. Section V presents experimental results, Section VI discusses some experience on analyzing ranking structures of communication fabrics, Section VII outlines related work, and Section VIII concludes the paper.

II. GENERAL INTUITION FOR RANKING STRUCTURES

Consider the state transition graph (STG) shown in Figure 2. It represents a system with four states $s_1$, $s_2$, $s_3$ and $s_4$ and five event-triggered transitions with events $e_1$, $e_2$, $e_3$, $e_4$ and $e_5$. We define four Boolean predicates $\pi_{s_1}$ through $\pi_{s_4}$ such that $\pi_{s_i}$ is true iff the system is in state $s_i$ for $i \in \{1, 2, 3, 4\}$. Due to acyclicity of the STG, once the system moves out of state $s_1$, it cannot go back to $s_1$. Therefore, once predicate $\pi_{s_1}$ evaluates to false, it cannot become true later. Paraphrasing this argument, we say that once $\beta_1 := \neg \pi_{s_1}$ becomes true, it remains true forever. Similarly, once the system leaves state $s_3$, it cannot go back to either $s_1$ or $s_2$. Hence $\beta_2 := \neg \pi_{s_1} \land \neg \pi_{s_3}$ also remains true once it becomes true. The same argument holds for $\beta_3 := \neg \pi_{s_1} \land \neg \pi_{s_3} \land \neg \pi_{s_2}$.

![DAG for a general partial order showing the barriers](image)

**TABLE I**

<table>
<thead>
<tr>
<th>Barriers</th>
<th>Barriers in disjunctive form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta_1 := \neg \pi_{s_1}$</td>
<td>$\Rightarrow \beta_1 \equiv \pi_{s_2} \lor \pi_{s_3} \lor \pi_{s_4}$</td>
</tr>
<tr>
<td>$\beta_2 := \neg \pi_{s_1} \land \neg \pi_{s_3}$</td>
<td>$\Rightarrow \beta_2 \equiv \neg (\pi_{s_1} \lor \pi_{s_3}) \equiv \pi_{s_2} \lor \pi_{s_4}$</td>
</tr>
<tr>
<td>$\beta_3 := \neg \pi_{s_1} \land \neg \pi_{s_2} \land \neg \pi_{s_3}$</td>
<td>$\Rightarrow \beta_3 \equiv \neg (\pi_{s_1} \lor \pi_{s_2} \lor \pi_{s_3}) \equiv \pi_{s_4}$</td>
</tr>
</tbody>
</table>
We say that $\beta_1, \beta_2, \beta_3$ are stabilizing assertions or barriers. An assertion $p$ is called stabilizing if $p$ eventually becomes true and remains true forever. In other words, $p$ is stabilizing if the LTL property $FGp$ holds. A stabilizing assertion $b$ is called a barrier if once $b$ becomes true, it remains true forever. Thus a barrier $b$ satisfies the LTL property $G(b \Rightarrow Xb)$. Clearly, all barriers are stabilizing assertions, but not all stabilizing assertions are barriers. For the above DAG, we note that the barriers $\beta_i$’s have a particular disjunctive characterization as shown in Table I. $\beta_i$’s represent various bi-partitions of the topologically sorted list of states of the DAG. Since each such bi-partition can be represented by a disjunction of some of the state predicates (i.e. union of states) in one of the partitions, formulations of $\beta_i$’s become inherently disjunctive. For example, $\beta_1$ represents the bi-partition $\langle s_1, s_3 \rangle \langle s_2, s_4 \rangle$ of the topologically sorted order $\langle s_1, s_3, s_2, s_4 \rangle$. Hence, we can use various stabilizing assertions like $\neg p_1$, or $p_2 \lor p_3 \lor p_4$ to represent barrier $\beta_1$. When a stabilizing assertion is represented using a disjunction of signals, it is called a disjunctive stabilizing assertion. A DAG structure of an STG naturally produces various disjunctive stabilizing assertions like $\beta_1, \beta_2$ and $\beta_3$ of Figure 2. A DAG of finitely many states naturally yields a well-founded ordering i.e. a ranking structure. A topologically sorted list of states of a DAG with $n$ nodes can produce $n - 1$ barriers. A list of all $n - 1$ barriers completely specifies the ranking structure of the DAG. A list with less barriers incompletely specifies the ranking structure. For example, the list $\langle \beta_1, \beta_2, \beta_3 \rangle$ completely specifies the ranking structure associated with the DAG of Figure 2. Conceptually, the terms ‘well-founded order’ and ‘ranking structure’ are the same, but ‘ranking structure’ has a more structural sense given by a list of barriers, while a well-founded order has a mathematically precise relational definition.

A. Identification of a barrier

In order to verify if a signal $b$ is a stabilizing assertion, we need a liveness verification engine that can prove/disprove a property of the form $FGb$. However, we only need a safety verification engine to demonstrate if a signal $b$ is a barrier. For such a candidate signal $b$, we need to create a monitor for the safety property $G(b \Rightarrow Xb)$ and invoke a safety verifier on this. Hence, a barrier can be verified more efficiently than a general stabilizing assertion. A complete reachability analysis algorithm like IC3 [12] or an incomplete algorithm like induction can be used to test if $G(b \Rightarrow Xb)$ holds. It has been observed that induction is particularly effective for proving this kind of property on certain benchmarks [9].

We say that $\beta_1, \beta_2, \beta_3$ are stabilizing assertions or barriers. An assertion $p$ is called stabilizing if $p$ eventually becomes true and remains true forever. In other words, $p$ is stabilizing if the LTL property $FGp$ holds. A stabilizing assertion $b$ is called a barrier if once $b$ becomes true, it remains true forever. Thus a barrier $b$ satisfies the LTL property $G(b \Rightarrow Xb)$. Clearly, all barriers are stabilizing assertions, but not all stabilizing assertions are barriers. For the above DAG, we note that the barriers $\beta_i$’s have a particular disjunctive characterization as shown in Table I. $\beta_i$’s represent various bi-partitions of the topologically sorted list of states of the DAG. Since each such bi-partition can be represented by a disjunction of some of the state predicates (i.e. union of states) in one of the partitions, formulations of $\beta_i$’s become inherently disjunctive. For example, $\beta_1$ represents the bi-partition $\langle s_1, s_3 \rangle \langle s_2, s_4 \rangle$ of the topologically sorted order $\langle s_1, s_3, s_2, s_4 \rangle$. Hence, we can use various stabilizing assertions like $\neg p_1$, or $p_2 \lor p_3 \lor p_4$ to represent barrier $\beta_1$. When a stabilizing assertion is represented using a disjunction of signals, it is called a disjunctive stabilizing assertion. A DAG structure of an STG naturally produces various disjunctive stabilizing assertions like $\beta_1, \beta_2$ and $\beta_3$ of Figure 2. A DAG of finitely many states naturally yields a well-founded ordering i.e. a ranking structure. A topologically sorted list of states of a DAG with $n$ nodes can produce $n - 1$ barriers. A list of all $n - 1$ barriers completely specifies the ranking structure of the DAG. A list with less barriers incompletely specifies the ranking structure. For example, the list $\langle \beta_1, \beta_2, \beta_3 \rangle$ completely specifies the ranking structure associated with the DAG of Figure 2. Conceptually, the terms ‘well-founded order’ and ‘ranking structure’ are the same, but ‘ranking structure’ has a more structural sense given by a list of barriers, while a well-founded order has a mathematically precise relational definition.

A. Identification of a barrier

In order to verify if a signal $b$ is a stabilizing assertion, we need a liveness verification engine that can prove/disprove a property of the form $FGb$. However, we only need a safety verification engine to demonstrate if a signal $b$ is a barrier. For such a candidate signal $b$, we need to create a monitor for the safety property $G(b \Rightarrow Xb)$ and invoke a safety verifier on this. Hence, a barrier can be verified more efficiently than a general stabilizing assertion. A complete reachability analysis algorithm like IC3 [12] or an incomplete algorithm like induction can be used to test if $G(b \Rightarrow Xb)$ holds. It has been observed that induction is particularly effective for proving this kind of property on certain benchmarks [9]. However, we used IC3-style algorithms instead of induction in our experiments. More discussion on this is deferred to Section V.

B. Utility of barriers in verification

Suppose that the system of Figure 2 stutters in a state until an event arrives that enables some of its outgoing transitions. We assume that all events $e_1, e_2, e_3, e_4$ and $e_5$ are guaranteed to always occur in the future i.e. the events are fair. We want to prove that if the system is not in $s_4$, it will eventually reach there. In LTL, this is expressed as $G(\neg s_4 \Rightarrow F\neg s_4)$. It turns out that the knowledge that the events are fair and that the state space has barriers $\beta_1, \beta_2$ and $\beta_3$ is sufficient to prove the liveness property $G(\neg s_4 \Rightarrow F\neg s_4)$, even without invoking a native liveness verifier. For any $\neg s_4$-state (i.e. $s_1$, $s_2$ or $s_3$) at least one event is guaranteed to occur in the future which causes the relevant barrier to switch so the system will not come back to any of the already visited $\neg s_4$-states. Since there are only finitely many $\neg s_4$-states, with the help of the fair events, the system will eventually arrive at $s_4$ and the provable existence of barriers is a certificate for this behavior.

We will demonstrate that the barriers can accelerate the response verification for communication fabrics. In the next section, we demonstrate how to combine some carefully chosen state predicates into barriers and how to use these to create an abstract DAG structure on the state space. Of course, the actual STG of a communication fabric is hardly a DAG. But if the design is free from a response bug, then we can make a “response-property specific DAG abstraction” of the STG. This abstraction is conceptual and modeled using implicit constraints; no explicit abstract STG is created in our algorithm. These concepts are illustrated with the help of the running example $\forall C$ of Figure 1 in the following sections.

III. Analysis of $\forall C$

When a request arrives on channel $a_1$ of $\forall C$ of Figure 1, i.e. when $a_1.req$ is asserted, $a_1.gnt$ may not be available immediately. We use predicate $G$ (for goal) to denote all states in the state space of $\forall C$ where $a_1.gnt$ is asserted. Dually, the states where $a_1.gnt$ is not asserted are represented by $\neg G$. If $\forall C$ is in $G$ when $a_1.req$ arrives, $\phi(a_1)$ holds immediately. However, if $\forall C$ is in $\neg G$, it needs be shown that $\forall C$ eventually will make progress to $G$. If this is the case, then $\phi(a_1)$ holds on $\forall C$ no matter which state $\forall C$ is in when $a_1.req$ arrives. It turns out that the $\neg G$ region of the state space of $\forall C$ can be clustered into three disjoint sub-regions $\neg G \land \sigma_1, \neg G \land \sigma_2$ and $\neg G \land \sigma_3$ where $\sigma_1 = Empty(B_1) \land Full(B_3), \sigma_2 = Empty(B_1) \land \neg Full(B_3)$ and $\sigma_3 = \neg Empty(B_1) \land \neg Full(B_3)$. Here $Empty(B)$ and $Full(B)$ are two predicates defined for each buffer $B$ such that $Empty(B) = 1$ if buffer $B$ is empty and $Full(B) = 1$ iff buffer $B$ is full. $\neg G \land \sigma_1, \neg G \land \sigma_2$, and $\neg G \land \sigma_3$ are pairwise disjoint, and $(\neg G \land \sigma_1) \lor (\neg G \land \sigma_2) \lor (\neg G \land \sigma_3) = \neg G$. An incompletely specified transition relation among these clusters is shown in Figure 3(a). The directed arrows represent how $\forall C$ travels from one cluster to another. The labels on the transitions are called fairness events and are explained below.

![Clusters in state space of $\forall C$](image-url)
The transition relation of Figure 3(a) is incompletely specified in that it highlights only those elements of the transition relation of \( V \choose C \) that are critical for proving \( \phi(a_1) \) and omits other details. The following two aspects are critical:

1. **The absence of inter-cluster loops in \( \neg G \):** Other than the self-loops on \( a_1 \) and \( a_2 \) there is no inter-cluster loop in the \( \neg G \) region. Once in \( a_3 \), returning to \( a_1 \) is not possible unless \( G \) is visited. Similarly, once in \( a_3 \), \( a_1 \) or \( a_2 \) cannot be revisited without visiting \( G \). Absence of inter-cluster loops in the \( \neg G \) region and transitions \( \{ a_1 \rightarrow a_2, a_2 \rightarrow a_3, a_2 \rightarrow G, a_3 \rightarrow G \} \) are the main reasons why \( \phi(a_1) \) holds. If there were an inter-cluster loops in \( \neg G \) and \( a_1, \text{req} \) arrives when \( V \choose C \) is in a state which is in such a loop, it may get trapped and would never be able to reach \( G \) from that state which would lead to violation of \( \phi(a_1) \). Note that self-loops on \( a_1 \) and \( a_2 \) would not create any problem in reaching \( G \) due to the fairness events as explained next.

2. **Fairness Events:** \( F_1 \) and \( F_2 \) are fairness events. \( F_1 \) represents the event when both the signals \( j_1, \text{gnt} \) and \( m_1, \text{gnt} \) are asserted. \( F_2 \) represents the event when \( i_1, \text{req} \) is asserted. The labels on the self-loop on \( a_1 \) and on the transition \( a_1 \rightarrow a_2 \) has the following meaning: when \( V \choose C \) is in some state in cluster \( a_1 \) and event \( F_1 \) arrives, it makes a transition to \( a_2 \), otherwise (denoted by \( \neg F_1 \)) \( V \choose C \) remains in \( a_1 \). States inside \( a_1 \) may form loops, called intra-cluster loops. Interestingly, all states in \( a_1 \) react to event \( F_1 \) so that \( V \choose C \) leaves \( a_1 \) once \( F_1 \) arrives. Event \( F_1 \) is guaranteed to arrive in the future due to the fairness assumption \( GF(j_1, \text{gnt}) \land GF(m_1, \text{gnt}) \) and the persistence property of xMAS fabrics [3]. Hence \( V \choose C \) is guaranteed not to get trapped forever in any intra-cluster loop inside \( a_1 \). Labels on the self-loop on \( a_2 \) and on the transitions \( a_2 \rightarrow a_3, a_2 \rightarrow G \) have a similar interpretation.

The fairness events and the absence of inter-cluster loops help \( V \choose C \) move gradually toward \( G \) from any state in \( \neg G \). We note that there are transitions from \( G \) to \( \{ G, \neg G \} \) in the state space. They are shown with broken arrows in Figure 3(a). The broken arrows are not crucial for proving \( \phi(a_1) \), while the solid arrows are. We are only interested in how \( V \choose C \) moves from a \( \neg G \)-state to \( G \). What happens after \( V \choose C \) arrives at \( G \) is immaterial. Hence, Figure 3(b) captures the core structure of the transition relation of \( V \choose C \) that are needed for proving \( \phi(a_1) \). The structure of Figure 3(b) is the DAG structure abstraction of the state space of \( V \choose C \) as discussed in Section II. The actual state space is not a DAG as evident from the broken arrows in Figure 3(a) (also, the clusters \( a_1 \)'s contain strongly connected components not shown in Figure 3(a)). In our case studies, we found that the existence of such an abstract DAG applies to any other communication fabric that is free from a response bug. Below we discuss how to construct this abstraction process implicitly for an arbitrary communication fabric using disjunctive stabilizing assertions, avoiding the creation of an explicit DAG.

### A. Barriers in \( V \choose C \)

Since the fairness of events is assumed as a part of the specification of \( \phi(\cdot) \), in order to prove \( \phi(\cdot) \) barriers \( \beta_i \)’s need to be found and associated with the abstract DAG of \( V \choose C \). If an implementation of \( V \choose C \) has Boolean signals corresponding to \( a_1, a_2, a_3 \) and \( G \), the following LTL (safety) properties can be hypothesized and proved on \( V \choose C \):

\[
\neg G \land X(\neg G) \land a_3 \Rightarrow X(a_3)
\]

\[
\neg G \land X(\neg G) \land (a_2 \lor a_3) \Rightarrow X(a_2 \lor a_3)
\]

In other words, signals \( a_3 \) and \( a_2 \lor a_3 \) are barriers in the \( \neg G \) region. Signals \( a_3 \) and \( a_2 \lor a_3 \) split \( \neg G \) into three implicitly ranked layers. The notion of ranking stems from the relative distance of an individual layer from the goal region \( G \). The disjunctive stabilizing behavior of signals \( a_3 \) and \( a_2 \lor a_3 \) capture this ranking information in an indirect way. Although these barriers were identified manually in the above example, below an algorithm is proposed that mines potential barriers automatically from bit-level netlist descriptions of xMAS fabrics.

### B. xMAS-specific predicates and the barrier mining algorithm

For xMAS fabrics, the information whether a FIFO buffer is empty or not, and whether a buffer is full or not has a natural connection to how a fabric eventually issues grant to a request. However in a bit-level version of the fabric, this information is not easily discernible. A dedicated algorithm would be required to identify appropriate logic (associated with a particular FIFO buffer) or to synthesize appropriate Boolean functions corresponding to the information required. Without any prior hint, it would be challenging for a general purpose liveness algorithm like \( k \)-LIVENESS to find such specialized information. Therefore, we introduce this information as explicit predicate signals in the design. In particular, for each buffer \( B \), predicates \( Empty(B) \) and \( Full(B) \) are introduced. During our experiments, before Algorithm 2 is invoked to discover stabilizing constraints, we initialize the set of candidate signals \( C \) as \( \{ Empty(B), \neg Empty(B), Full(B), \neg Full(B) | B \) is a buffer in the fabric \}. Using the set \( C \) of auxiliary predicates, our mining algorithm works as follows:

#### Barrier Mining Algorithm:

For each signal \( c \in C \), construct a monitor for the property \( G(c \Rightarrow Xc) \). It is done by introducing a register \( r_c \) for each such signal \( c \in C \) where \( c = input(r_c) \), and an implication logic \( output(r_c) \Rightarrow c \). A safety verifier is used to filter out the signals in \( C \) whose monitors fail this safety test. Then barriers from the set \( C^2 \) are tested using the same monitor construction approach. The signals in sets \( C \) and \( C^2 \) which pass the test are collected and used in the subsequent liveness algorithm. The overall approach is similar to that of finding stabilizing signals as described in [9]. Based on our insights into the xMAS structures, here we focus only on the set \( C \) and \( C^2 \) for candidate barriers instead of all gate outputs and register outputs as recommended in [9]. The runtime of this mining process is reported in Section V.

The following remarks explain a few of the xMAS specific optimizations adopted in the barrier mining algorithm:
(1) although Sections II and III have identified barriers as disjunctions of predicates representing states or state clusters, it is possible to achieve further simplifications in barrier descriptions for XMAS fabrics. For example, we saw that \( \sigma_3 \) and \( \sigma_2 \lor \sigma_3 \) describe all necessary barriers in \( \neg G \) in \( \forall C \). Closer observation reveals that when the capacities of buffer \( B_1, B_2 \) and \( B_3 \) are equal, \( \neg Empty(B_1) \Rightarrow \neg Full(B_3) \). This can be formulated as a safety lemma and proved on \( \forall C \) separately. Consequently, \( \sigma_3 \) can be described as \( \neg Empty(B_1) \). On the other hand, \( \sigma_2 \lor \sigma_3 \) can be simplified to \( \neg Full(B_3) \) as follows: \( \sigma_2 \lor \sigma_3 \equiv (\neg Empty(B_1) \land \neg Full(B_3)) \lor (\neg Empty(B_1) \land \neg Full(B_3)) \equiv \neg Full(B_3) \). Thus, although \( \sigma_1 \)'s are defined using conjunctions of signals from \( C \), the set \( C \) itself is a rich collection of candidate signals that can act as barriers. Searching only in \( C \) also avoids the combinatorial blow-up involved in considering conjunctions of signals in \( C \) as candidates barriers. (2) The ranking structures of the credit-based flow-control systems studied so far have the interesting behavior that their state clusters exhibit linear flow of control as shown in Figure 3(b) (i.e. their DAG structures are essentially path-like, not tree-like or general DAG-like).

For such fabrics, these clusters have the following generic description: \( c_1, \neg c_1 \land c_2, \neg c_1 \land \neg c_2 \land c_3 \) etc. where \( c_i \)'s are predicates in \( C \). For example, in \( \forall C \), the clusters are \( \neg Empty(B_1), Empty(B_1) \land \neg Full(B_3) \) and \( Empty(B_1) \land Full(B_3) \). This pattern simplifies the description of the associated barriers as shown in Table II. This motivates searching for barriers directly in the sets \( C, C'^2, C'^3 \) etc. where \( C'^n = \{ c_1 \lor c_2 \lor \ldots \lor c_n : c_i \in C, \forall i \in \{1, 2, \ldots, n\} \} \). These disjunctive descriptions of \( C'^n \)'s give another motivation for extending the notion of candidate stabilizing constraints, from simple AIG signals (as proposed in the basic \( k\)-liveness algorithm) to disjunctions of AIG signals. As another heuristic optimization in our experiments, candidates beyond \( C \) and \( C'^2 \) were not considered because it was suspected that the benchmarks studied did not have a `deep' partial order associated with their state spaces. This avoids another combinatorial blow-up associated with considering \( C'^n \)'s for large \( n \). However, based on the available computing resources, our algorithm can be parameterized easily for searching in various sets \( C'^n \)'s. Our experiments showed that acceptable run-times were obtained for the benchmarks even when barriers were constructed only from signals in \( C \) and \( C'^2 \). For more challenging benchmarks, searching in \( C'^n \)'s for \( n > 2 \) may be justified.

### Table II
**Generic barrier descriptions and their simplifications for credit-based flow control systems**

<table>
<thead>
<tr>
<th>barrier</th>
<th>description</th>
<th>equivalent simpler description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \beta_1 )</td>
<td>( c_1 \lor c_2 \lor \neg c_1 \land \neg c_2 \lor c_3 )</td>
<td>( c_1 \lor c_2 \lor c_3 )</td>
</tr>
<tr>
<td>( \beta_2 )</td>
<td>( c_1 \lor (\neg c_1 \land \neg c_2 ) \lor (\neg c_1 \land \neg c_2 \land c_3 ) )</td>
<td>( c_1 \lor (\neg c_1 \land \neg c_2 \land c_3 ) )</td>
</tr>
</tbody>
</table>

Below we explain how these stabilizing assertions can be used to speed up a conventional liveness algorithm. In our experiments, \( k\)-LIVENESS was used as the native liveness algorithm. However, it is possible that stabilizing assertions can accelerate other off-the-shelf algorithms based on BDD constructions or liveness-to-safety transformations. Whether the ranking structures of communication fabrics can accelerate these algorithms and which algorithm would be the most effective are interesting research questions left to the future.

### IV. \( k\)-LIVENESS ALGORITHM FOR RESPONSE VERIFICATION

As an \( \omega \)-regular property, the response property \( \phi(a_1) \) can be model-checked by converting the problem into a property of the form \( \text{FG} \phi \) using generalized B"uchi automata construction. However, the core idea of \( k\)-LIVENESS might be able to prove \( \phi(a_1) \) in a more direct way that avoids the complicated B"uchi construction. The advantage of our property specific construction, lies in its easy implementation; it has more of a software engineering benefit than any technical novelty. Whether our construction has any fundamental efficiency benefit over the generalized B"uchi automata approach is left to the future.

Below we illustrate our property specific construction for verifying \( \phi(a_1) \) on \( \forall C \). The idea can be generalized for any \( \phi() \) and for any other bit-level sequential netlist.

**Intuition:** If \( \phi(a_1) \) is to hold on \( \forall C \), i.e. if every \( a_1, \text{req} \) is eventually followed by an \( a_1, \text{gnt} \), the interval for which \( \forall C \) waits for \( a_1, \text{gnt} \) once \( a_1, \text{req} \) is asserted (called the **pending interval**) can never be infinite. Since the fairness signals help \( \forall C \) to produce a grant, an equivalent argument is that the number of times each fairness signal is asserted during any pending interval can never be infinite. Thus, the \( k\)-LIVENESS algorithm can be used to prove that the number of times fairness signals are asserted during a pending interval is finite for \( \forall C \).

**Monitor construction:** A monitor that implements the above idea can be constructed as shown in Algorithm 1. This monitor is attached to \( \forall C \), and \( k\)-LIVENESS is called to prove that signal \text{allFairCount} can assume logic value 0 only finitely many times. Algorithm 1 is written in an imperative pseudocode with standard semantics where all assignments are valuated in parallel at every clock. Verilog-style semantics are used for the data-types, namely, \text{wire} represents combinational signals and \text{reg} represents sequential signals (i.e. registers). All \text{reg} variables are initialized to logic 0 and for a \text{reg} variable \( r \), \text{next}(r) \) computes its next state value.

Algorithm 1 demonstrates how the core idea of \( k\)-LIVENESS can be used to verify \( \phi(a_1) \) in a direct way, without constructing a generalized B"uchi automaton for the whole property \( \phi(a_1) \). However, this basic monitor of Algorithm 1 does not leverage the stabilizing constraints introduced in Section III. Below we show how Algorithm 2 modifies Algorithm 1 to include the influence of the stabilizing constraints. In Algorithm 2, \( C \) stands for a set of stabilizing constraints supplied to the main algorithm (either mined automatically, say using our barrier mining algorithm, or supplied by the designer). The roles of stabilizing constraints is captured in the variable \text{arenaViolation}.
Algorithm 1: $k$-Liveness Monitor for Proving $\phi$ on $\forall \mathbf{C}$

1. wire pending, pendingInterval, allFair;
2. wire fair[6] =
   \{i1.req, i2.req, j1.gnt, j2.gnt, m1.gnt, m2.gnt\};
3. reg oracleSaved, gntSaved, fairFlop[6];
4. pending := a1.req $\land$ $\neg$a1.gnt;
5. next(oracleSaved) :=
   oracleSaved $\lor$ (oracle $\land$$\neg$oracleSaved $\land$ pending);
6. next(gntSaved) := gntSaved $\lor$ (oracleSaved $\land$ a1.gnt);
7. pendingInterval := oracleSaved $\land$$\neg$gntSaved;

\[
allFair := \bigwedge_{i=1}^{6} fairFlop[i];
\]

8. for $i \in \{1, 2, \ldots, 6\}$ do
   9. if allFair then
      10. next(fairFlop[i]) := 0;
   11. else
      12. next(fairFlop[i]) := fairFlop[i] $\lor$ fair[i];

13. allFairCount := $\neg$(pendingInterval $\land$ allFair);

Algorithm 2: Arena-aware $k$-Liveness Monitor

1. wire pending, pendingInterval, allFair;
2. wire fair[6] =
   \{i1.req, i2.req, j1.gnt, j2.gnt, m1.gnt, m2.gnt\};
3. reg $f_1$, $f_2$, $\ldots$, $f_C$;
4. reg oracleSaved, gntSaved, fairFlop[6];
5. for all $c \in C$ do
   6. if oracle $\land$$\neg$oracleSaved $\land$ pending then
      7. next($f_c$) := $c$;
   8. else
      9. next($f_c$) := $f_c$;

10. arenaViolation := $\bigvee_{c \in C} (f_c \neq c)$;
11. pending := a1.req $\land$$\neg$a1.gnt;
12. next(oracleSaved) :=
   oracleSaved $\lor$ (oracle $\land$$\neg$oracleSaved $\land$ pending);
13. next(gntSaved) :=
   gntSaved $\lor$ (oracleSaved $\land$(a1.gnt $\lor$ arenaViolation));
14. pendingInterval := oracleSaved $\land$$\neg$gntSaved;

\[
allFair := \bigwedge_{i=1}^{6} fairFlop[i];
\]

15. for $i \in \{1, 2, \ldots, 6\}$ do
   16. if allFair then
      17. next(fairFlop[i]) := 0;
   18. else
      19. next(fairFlop[i]) := fairFlop[i] $\lor$ fair[i];

20. allFairCount := $\neg$(pendingInterval $\land$ allFair);

A. Working principle and correctness of the algorithms

In both Algorithms 1 and 2, a primary input oracle is used to model the non-deterministic choice for an arbitrary pendning interval. It triggers the monitor at an arbitrary time step when $a_1$.req is asserted, but $a_1$.gnt is not. Once this event occurs, register oracleSaved remembers the event and disables the oracle forever. This event also starts the pending interval under examination. This interval is ended in Algorithm 1 when a subsequent $a_1$.gnt arrives. Note that this construction works because there is no obligation of matching an $a_1$.gnt with a corresponding $a_1$.req and $a_1$.req has the persistence property [3]. Algorithm 2 differs from Algorithm 1 in the way it closes the pending interval. When it opens the pending interval, it takes a snap-shot of all signals of the form $a_1 \lor \ldots \lor a_c$ in registers $\{f_c\}$ where signals $\forall_i a_i$ are disjunctive stabilizing constraints discovered in the pre-processing phase. Then using variable arenaViolation, it keeps track of whether any one such signal has changed its value (i.e. the check $f_c \neq c$). Since each such signal cannot flip its value within a potential counter-example to the response property, a scenario under which it changes its value cannot be a candidate for violation to $\phi$. If any such signal has changed its value, then the algorithm closes the pending interval (through $a_1$.gnt $\lor$ arenaViolation). Note that Algorithm 2 achieves an early closure of the pending interval compared to Algorithm 1 due to the use of disjunctive stabilizing constraints. This helps in a subsequent reduction in the number of iterations in the proof phase. Clearly, these constructions will work for general response verification situations as well.

V. Experimental Results

We use ABC [11] as the verification environment for both safety and liveness. A Verilog implementation of the XMAS library was developed, closely following the logical definitions given in [1]. This was used to implement various models of communication fabrics. A tool VERIABC was used to bit-blast these Verilog models. VERIABC uses Verific, a commercial Verilog front-end analyzer, to read in the Verilog models. The benchmarks considered are industrially relevant and their designs are available in the literature ([3], [4], [1]). They represent the basic virtual channel ($\forall \mathbf{C}$), a virtual channel with channel buffer ($\forall \mathbf{C}$,$\mathbf{B}$), a virtual channel with ordering ($\forall \mathbf{C}$, $\mathbf{O}$) and a cascaded virtual channel with ordering ($\forall \mathbf{C}$, $\mathbf{C}$). The fabric $\forall \mathbf{C}$,$\mathbf{O}$ is a cascade of two instances of $\forall \mathbf{C}$.

The experiments reported in Table III were performed on a cluster computer with 4 Intel(R) Xeon(R) E5-2670 processors each with eight 2.6 GHz cores, each core having 20MB cache, and a total of 132GB RAM. For proving all safety obligations, we used property directed reachability [12] engine (command pdr) of ABC. Columns 1, 2 and 3 list the design name, number of primary inputs and number of flip-flops respectively. The other columns of Table III represent the following runs of experiments:

- column 4 (l2s) reports the run-time of solving $\phi(a_1)$ using the liveness-to-safety transformation [7].
TABLE III

<table>
<thead>
<tr>
<th>design</th>
<th>PT</th>
<th>t/f</th>
<th>L2s</th>
<th>kcs -c</th>
<th>kcs -C (pdr)</th>
<th>kcs -C (pdr -a)</th>
<th>kcs -g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PT</td>
<td>k</td>
<td>PPT</td>
<td>k</td>
<td>PPT</td>
<td>PT</td>
<td>k</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.12</td>
<td>2</td>
<td>1.96</td>
<td>0.11</td>
<td>1</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>77</td>
<td>5.92</td>
<td>2</td>
<td>77.29</td>
<td>1.73</td>
<td>1</td>
<td>6.19</td>
</tr>
<tr>
<td></td>
<td>182</td>
<td>6.32</td>
<td>4</td>
<td>7.03</td>
<td>5.06</td>
<td>2</td>
<td>1.45</td>
</tr>
</tbody>
</table>

- column 5-6 (kcs -c) reports the run-time of solving \( \phi(a_1) \) using k-LIVENESS. Command kcs invokes k-LIVENESS in ABC. Switch -c allows users to supply model specific invariants manually. We supplied a set of known design invariants [3] for our models during these experiments. They restrict the state space and help to get the proofs significantly faster. In this experiment, we use Algorithm 1 to verify \( \phi(a_1) \). Here \( k \) denotes the number of iteration kcs required to prove \( \phi(a_1) \) on various designs.

- column 7-9 (kcs -C) - here switch -C mines disjunctive stabilizing constraints automatically. If a signal \( a \) is suspected to be stabilizing, a new signal is created corresponding to \( G(a \Rightarrow Xa) \). For a set of suspect signals, a set of such property signals were created and pdr was invoked to check them one after another. ‘PPT’ denotes the pre-processing time - the time our miner took to discover the disjunctive stabilizing signals sequentially. ‘PT’ is the proof time - the time kcs took to solve \( \phi(a_1) \) using the stabilizing constraints discovered in the pre-processing phase, and \( k \) again denotes the number of iterations kcs required to prove \( \phi(a_1) \) on various designs.

- column 10-12 (kcs -C (pdr -a)) - In this experiment, candidate disjunctive stabilizing constraints are tested in one shot. The task of discovering disjunctive stabilizing constraints is framed as a multi-property safety verification problem and the command pdr is invoked in ABC using the switch -a. This new capability was enhanced recently in ABC to allow for solving multi-output properties efficiently. The implementation is based on a breath-first approach and per-property timeout. It can prove, disprove or time out individual properties. The runtime of the multi-output solver is much better than the cumulative runtime of solving each property separately. PPT, PT and \( k \) mean pre-processing time, proof time and the number of proof iterations as described above. Use of the same multi-property engine for solving the iterative proof phase also resulted in run-time improvement for most of the cases.

- column 13 (kcs -g): This switch attempts to prove \( \phi(a_1) \) with user supplied ranking structure. We manually derived ranking structures of the benchmarks and manually chose the disjunctive stabilizing constraints, generated by our mining algorithm, to match with the manually derived ranking structures. This set of experiments shows that once the ranking structure is accurately known, the tool can produce very efficient proofs of response.

All times reported are in seconds. One benefit of using disjunctive stabilizing constraints is reflected in columns 8 and 11; these numbers are now reduced compared to the corresponding numbers in column 5. This also reduces the run-time (column PT) of the proof step, sometimes quite significantly. For example, the total run-time for solving VCO with disjunctive constraints is 209.98 sec. + 522.25 sec. = 732.23 sec, which is a good reduction over the 957.77 sec required to solve the same benchmark without disjunctive constraints. This run-time greatly reduces to 30.87 sec. + 96.71 sec. = 127.58 sec. when the multi-property solver used. Since the use of disjunctive stabilizing constraints achieves the proof with a smaller \( k \), it puts less stress on the safety verification engine and increases the chances of convergence of the final proof for challenging benchmarks.

VI. OUR EXPERIENCE: WHAT WORKS, WHAT DOESN’T

Bit-level liveness verification of industrially relevant communication fabrics are hard because they contain deeply pipelined logic for multi-phase transactions, and ordering logic for several virtual channels and peer-to-peer traffic. Native liveness verification algorithms, such as L2S and k-LIVENESS with no enhancement, are only moderately successful on these benchmarks. The ranking structures discovered in the fabric state spaces are important invariants that can enhance such algorithms. To understand the monotone behavior of the fabrics, in addition to mining disjunctive stabilizing assertions automatically, we also constructed ranking structures for several communication fabrics manually. Our observations can be found in previous works [13]. We found that deriving ranking structures for a small fabric like VC is quite manageable. VEB and VEO were increasingly more challenging and for EVO, we only managed to do a partial analysis manually. We learned a few interesting points through this manual process: (1) it has a close connection to the if-then-else reasoning associated with the flit propagation logic inside a fabric. The ranking structures thus generated have a concise description. The mining algorithm presented in the paper yields only a collection of disjunctive stabilizing assertions, but these do not readily translate into a concise ranking structure. The advantage of having a concise ranking structure is reflected in the experiment with kcs -g command. The impressive run-time was a result of using the relevant stabilizing assertions obtained from the manually derived ranking structures. Constructing a high-level algorithm that can leverage the if-then-else reasoning of a fabric to produce a concise ranking structure is left for the future. (2) A fabric often inherits structural features from another fabric. Manual analysis of the latter often facilitates manual analysis of the former. For example, VEB and VEO are modifications of VC. Ranking structure analysis of VC was helpful for ranking structure analysis of...
\(\text{V\&E\&B} \text{ and } \text{V\&E\&O}\). (3) The notion of barriers as introduced in Section II is applicable to any partial order. Although the fabrics we analyzed all have a linear path-like structure for their partial order (as in Figure 3(b)), in general the partial order underlying a state space can have an arbitrary DAG structure. However, the abstract DAGs for communication fabrics probably are not arbitrarily complex and might have simple path-like or tree-like structures. (4) Fabrics are often constructed by composing smaller fabrics. We observed that in some cases the ranking structures of the smaller fabrics can be composed to generate the ranking structure of the larger fabric. Possibly some interesting compositional reasoning algorithms can be designed in this space. (5) We observed that even if a mining algorithm only discovers an incompletely specified ranking structure, it can still improve the run-time of the basic \(k\)-LIVENESS algorithm.

VII. RELATED WORK

Ranking-oriented proofs of liveness are not new. In the 1980’s, Manna and Pnueli pioneered this notion based on well-founded induction and ranking functions in proving liveness of general reactive systems [6]. Research interest in this area has been renewed recently, particularly in the context of termination analysis of software [14] [15] [16]. This research relies on developments in mathematical and algorithmic techniques for synthesizing ranking functions for program loops [17] [18]. However, no prior work has addressed ranking-oriented proofs of liveness for communication fabrics. The work by Manolios [19] reasons about ranking structures of industrially relevant hardware systems is not directly applicable to our cases. Our work seems to be the first to identify monotonicity in the operations of xMAS communication fabrics and this can be described most naturally as ranking structures as presented in the paper. Whether closed-form mathematical expressions can be constructed for such ranking structures as in [17] [18] is an open question. Another area, pertinent to progress analysis, is stability analysis of dynamical systems using Lyapunov functions. Recently, experts in formal methods and system theory have collaborated on symbolic modeling and verification of dynamical systems [20], on using Lyapunov analysis techniques for formal verification of software [21] and on analyzing continuity and robustness of software [22]. xMAS is a relatively new model of computation (MoC) and its connection to dynamical systems and imperative (or functional) programming is not well-understood yet. Whether xMAS models can benefit from the analytical techniques developed for dynamical systems analysis or from the algorithmic techniques developed for ranking function synthesis of software is left for future research.

Another major paradigm for modeling computation and communication is data-flow process networks (DFPN) [23]. Among different variants of DFPN, xMAS is mostly similar to Boolean Data Flow (BDF), though xMAS differs from DFPN in few aspects. While MoCs like Synchronous Data Flow (SDF) or Cyclo-static Data Flow (CDF) have interesting mathematical characterizations that can be leveraged while analyzing their safety and liveness properties, such characterizations do not apply to general BDF. It is known that BDF is Turing complete, which means that the general model checking question for BDF is undecidable. On the other hand, xMAS is a framework for modeling finite state systems, and model checking is obviously decidable for xMAS. It is not known whether the graph theoretic algorithms developed for analyzing less-expressive DFPN like SDF, CDF can be applied to general xMAS models, but certainly a bit-blasted xMAS model can be analyzed using hardware model checking techniques. In this paper, we have taken the hardware model checking approach for analyzing xMAS models. It remains to be seen whether the DFPN-like behavior of xMAS can be leveraged in general to speed up the verification process.

[24], [4] and [25] are among the recent works on liveness verification of xMAS fabrics, most closely related to our problem. Ray et al. solved it by splitting \(\phi(\cdot)\) into intermediate safety properties. While this technique scales well, its major drawback is the need for extensive manual analysis and understanding of the designs to generate the intermediate safety properties. In contrast, the algorithm proposed here for mining disjunctive stabilizing constraints needs no manual understanding of the design. It solves the same problem automatically with comparable scalability. In [4], Gotmanov et al. proposed an efficient compositional technique for proving deadlock freedom of communication fabrics by composing sets of sufficient conditions for deadlock freedom of the individual xMAS components. This system-wide condition is checked with a SAT solver for unsatisfiability. In [25], Schmaltz et al. proposed algorithms based on graph analysis that certifies deadlock freedom for networks-on-chip represented using xMAS. Possibly our work on response verification has subtle conceptual connections to the deadlock verification methods of [25] and [4], and may complement them by introducing the notion of ranking structure. Discovery of high level structure in system behavior is of independent interest. The monotone order discovered in this work or its possible variants may be used in performance analysis [26] or for general liveness verification of xMAS fabrics. The benchmarks targeted in [4] are different from those in [25]. Our benchmarks are more closely akin to those of [4]. In the future, we plan to try our technique on benchmarks used in [25] as well. Another difference with [25] is that their algorithm works at the micro-architecture level and not at the bit-level. The compositional technique in [4] works at the bit-level but it does not reveal any transaction-level information about the fabric operation as does the ranking structure.

Research on general algorithms for bit-level liveness verification has gained remarkable momentum recently. New algorithms and tools for hardware model checking are being developed. Biere et al. showed how algorithms for safety verification can be re-used for liveness through a liveness-to-safety conversion [7] [27]. Bradley et al. proposed FAIR [8], a scalable, incremental algorithm for liveness verification based on the very successful safety verification algorithm IC3 [12]. \(k\)-LIVENESS is a recent addition to the repertoire of scalable
liveness algorithms. Both $k$-LIVENESS and our methodology have strong conceptual connections with FAIR. The notion of ‘arena’ used in Algorithm 2 was inspired by FAIR.

VIII. CONCLUSION

We discussed how fairness events can induce ranking structures in the state spaces of communication fabrics. These have a close connection with a ranking-oriented proofs for liveness. We demonstrated how this can be leveraged for proving a response property. To utilize these ranking structures in an automated verification framework, we augmented the basic $k$-LIVENESS algorithm with the notion of disjunctive stabilizing constraints. We experimented with a collection of communication fabrics and discovered their ranking structure manually. Then we devised an algorithm that automatically mines disjunctive stabilizing constraints. We ran two sets of experiments, one using manually generated stabilizing constraints and the other using automatically generated stabilizing constraints. Experiments demonstrated the overall effectiveness of ranking structures in speeding up the verification process. Future work includes generalizing the idea of ranking structures, presented in this paper, to accommodate more general structures such as trees or DAGs. Also, a comparison of quality-of-result and run-time of our IC3-based barrier detection scheme vs. the induction based barrier detection scheme [9] on XMAS communication fabrics would be of interest.

ACKNOWLEDGEMENT

We thank Alan Mishchenko for many helpful discussions on ABC, Niklas Een for suggesting the alternative monitor construction for response properties, Jiang Long for help in bit-blasting our models, and anonymous reviewers for suggested improvements in the initial draft of the paper. This work has been supported in part by the SRC, NSA and industrial sponsors: Actel, Altera, Atrenta, Cadence, Calypto, IBM, Jasper, Magma, Oasys, Real Intent, Synopsys, Tabula, and Verific, with particular thanks to Intel for their initial sponsorship of this research.

REFERENCES


APPENDIX

The core observation that Intel’s researchers made while developing XMAS was that communication fabrics are essentially ‘networks of finite FIFO buffers, with intervening glue logic’. In most of the cases, this glue logic can be described with a small number of characteristic primitives. XMAS is, therefore, described as a library of a small number of structural components, viz. finite FIFO buffer, source and sink of data items, function block, synchronization primitives fork and join, decision primitives arbiter and switch.

Each primitive of XMAS is defined as a synchronous Boolean circuit. FIFO buffer, source, sink and arbiter have sequential components, while the rest are defined as combinational circuits. A communication fabric is constructed by
stitching instances of different components together. A fabric thus constructed becomes a synchronous, sequential Boolean circuit triggered by a single global clock. Its timing model follows the synchronous model of time as in [28]. For the ease of high-level modeling and informal exchange of designs, each component is represented with a visual symbol as shown in Figure 4. Logical definition of individual components are presented below. For further detail, see [1].

Fig. 4. XMAS symbols for structural components of communication fabrics

**FIFO buffer:** Consider a synchronous FIFO queue with a standard interface comprising a read port and a write port as shown in Figure 5. The queue has two parameters: size \( k \) (the number of elements it can contain) and a type \( \tau \) (the type of elements it can contain). To compose two instances of such a queue back-to-back without any extra glue logic, XMAS proposes slightly modified interface signals for each FIFO buffer as described below. Figure 6 shows composition of two FIFO buffers with such a modified interface definition.

![FIFO buffer diagram](image)

**Function:** A function is an XMAS primitive that transforms data. It is parameterized by two types \( \alpha \) and \( \beta \) and a function \( f : \alpha \rightarrow \beta \). It has an input port \( i \) : \( \alpha \) and an output port \( o \) : \( \beta \). It is characterized by the following combinational equations: \( o \cdot \text{irdy} := i \cdot \text{irdy}, o \cdot \text{data} := f(i \cdot \text{data}), i \cdot \text{trdy} := o \cdot \text{trdy} \) \( \land \) \( \neg o \cdot \text{irdy} \).

**Join:** A join is the dual of a fork. It has two input ports \( a \) : \( \alpha \) and \( b \) : \( \beta \) and one output port \( o \) : \( \gamma \). It is parameterized by a single function \( h : \alpha \times \beta \rightarrow \gamma \). Intuitively, a join takes two input packets (one at each input) and produces a single output packet. It coordinates the input and outputs so that a transfer only takes place when the inputs are ready to send and the output is ready to receive. Formally, \( o \cdot \text{trdy} := o \cdot \text{trdy} \land b \cdot \text{trdy}, o \cdot \text{data} := f(i \cdot \text{data}), i \cdot \text{trdy} := o \cdot \text{trdy} \land h(a \cdot \text{data}, b \cdot \text{data}) \).

**Fork:** A fork is a primitive with one input port \( i \) : \( \alpha \) and two output ports \( a \) : \( \beta \) and \( b \) : \( \gamma \) parameterized by two functions \( f : \alpha \rightarrow \beta \) and \( g : \alpha \rightarrow \gamma \). Intuitively, a fork takes an input packet and creates a packet at each output. It coordinates the input and outputs so that a transfer only takes place when the input is ready to send and both the outputs are ready to receive. Formally, \( a \cdot \text{irdy} := i \cdot \text{irdy} \land h \cdot \text{trdy}, a \cdot \text{data} := f(i \cdot \text{data}), b \cdot \text{irdy} := i \cdot \text{irdy} \land \neg a \cdot \text{trdy}, b \cdot \text{data} := g(i \cdot \text{data}) \) and \( i \cdot \text{trdy} := a \cdot \text{trdy} \land b \cdot \text{trdy} \).

** Arbitration:** Arbitration is modeled by a merge primitive that selects one packet among multiple input ports and one output port. Requests for a shared resource are modeled by sending packets to a merge and a grant is modeled by the selected packet. A complete definition of a two-input merge that has two input ports \( i \) : \( \alpha \) and \( b \) : \( \beta \) and one output port \( o \) : \( \alpha \)

\[ u := 1 \text{ if } a \cdot \text{irdy} \land \neg b \cdot \text{irdy} \]
\[ 0 \text{ if } \neg a \cdot \text{irdy} \land b \cdot \text{irdy} \]
\[ \neg \text{pre}(u) \text{ if } \text{pre}(u) \land \neg o \cdot \text{trdy} \]
\[ \text{pre}(u) \text{ otherwise} \]

**Persistence Property of XMAS Components:** XMAS components are so defined that the associated channels hold a property called persistence. Informally, a signal is persistent means that once the signal is asserted by the initiator agent, it remains asserted until it is properly served by the target agent. For any XMAS channel \( u \), both its irdy and trdy signals are persistent. Persistence of these two signals of channel \( u \) is termed as forward persistence and backward persistence and denoted with \( \text{FwdPersistence}(u) \) and \( \text{BwdPersistence}(u) \) respectively. Formally, these two properties of channel \( u \) can be defined in LTL as follows:

\[ \text{FwdPersistence}(u) \equiv G((u \cdot \text{irdy} \rightarrow u \cdot \text{trdy}) \Rightarrow Xu \cdot \text{irdy}) \]
\[ \text{BwdPersistence}(u) \equiv G((u \cdot \text{trdy} \rightarrow u \cdot \text{trdy}) \Rightarrow Xu \cdot \text{trdy}) \]

Persistence is an important property that every XMAS channel satisfies. It greatly simplifies behavior of XMAS fabrics, significantly limits potential deadlocks and allows for simpler definition of channel liveness. It helps in compositional analysis of fabric behaviors and their liveness analysis with intermediate safety properties.