Abstract—We present a liveness verification framework that extends $k$-Liveness algorithm with disjunctive stabilization constraints. We demonstrate effectiveness of our algorithm for verification of response property for communication fabrics. We also provide an open-source implementation of our algorithm in the ABC framework. This framework can be used as a general model checker for liveness properties on any sequential circuit. Experiments are performed on industrially relevant public benchmarks, and preliminary results are very encouraging.

I. INTRODUCTION

Liveness properties offer a neat, and elegant way of specifying reactive behaviors like something ‘good’ will eventually happen. This way, one can write end-to-end specifications without worrying too much about the internal details of a design. Unfortunately, traditional model checking algorithms that verify such specifications involve computationally expensive formulations, and suffer from scalability problems. Typically, these algorithms attempt to enumerate all strongly connected components (SCC) of state transition graphs. This way, they try to rule out existence of any counterexample to the liveness property, but end up being unscalable for large designs.

It has been observed in prior research, however, that real-life designs often carry intermediate hints that may be used to construct deductive proof of liveness without invoking SCC analysis. With a little abuse of terminology, we collectively call this alternative hint-based approach as ranking-oriented approach for proving liveness. These proofs are usually scalable compared to their SCC-oriented counterparts. On the down side, it is very challenging to devise a completely automated, scalable proof engine that will discover necessary hints, and eventually construct a proof of liveness for an arbitrary system. Within the last couple of years, however, we have witnessed major advances in bit-level liveness verification algorithms. The most remarkable are two algorithms, namely, FAIR [7], and $k$-LIVENESS [10]. Intriguingly, both of the algorithms attempt to mine such intermediate hints (a.k.a. invariants) automatically from the design, and leverage these hints to simplify their proof obligations. Before these two algorithms, we had no effective tool for discovering ranking-oriented information from general hardware designs. In a sense, these new algorithms hold promises of bridging the gap between automated liveness analyses in SCC-oriented, and ranking-oriented ways.

Our specific interest lies in proving liveness properties of communication fabrics. These are an integral part of modern hardware systems. They range from local circuitry like hardware scoreboards to system-wide network-on-chips. Being prone to liveness bugs like deadlocks, and livelocks, these fabrics require rigorous formal verification at every layer of abstraction during their design. Surprisingly, a human designer can produce informal proofs of liveness of such fabrics very quickly at the micro-architecture level. But current bit-level liveness verifiers take prohibitively long time to (re)produce the same proof after bit-blasting. We believe that this performance gap could be attributed to the blind enumeration of SCCs by the bit-level solvers. A ranking-oriented model checker, which can somehow mine the designer’s intuition, could potentially (re)produce the proof quickly, and scalably. Our quest is to develop an appropriate ranking-oriented model checker which can successfully execute this job.

Toward this goal, we show that a very natural subset of design signals of communication fabrics may be treated as candidate hints. Given such a set of candidate hints, we demonstrate that an augmented version of $k$-LIVENESS algorithm can successfully mine relevant information, and leverage them to produce a scalable proof. We focus on one particular liveness property called response [17]. It is one of the most common, and crucial liveness properties that a designer would like to prove on her fabric. A formal description of this property is given in Section II-C. Our experiments deal with a collection of industrially relevant benchmarks, and results demonstrate an order of magnitude speed up achieved by our algorithm compared to traditional model checkers. Our contributions in this paper are the following:

• We introduce the notion of disjunctive stabilizing constraints that generalizes the pre-processing step of $k$-LIVENESS algorithm. Informally, disjunctive stabilizing constraints can expose a hierarchy of monotone signals in the design, and we demonstrate how these constraints (hierarchical monotone signals, in other words) can simplify the subsequent liveness proof. Objective of the pre-processing step of $k$-LIVENESS is to reduce the number of iterations in the subsequent proof phase. We show that the notion of disjunctive stabilizing constraints achieves further reduction in the number of iterations compared to the original $k$-LIVENESS algorithm. See Section IV for details. The notion of disjunctive stabilizing constraints is inspired by our analysis of communication fabrics, and they have been used in proving response properties of the same in this work. However, the notion is general, and extends the general scope of the pre-processing step of $k$-LIVENESS algorithm for general hardware systems. In this paper, we will use terms disjunctive stabilizing constraints, and hierarchical monotone signals interchangeably.

• We identify a class of design predicates for communication fabrics that exhibits the hierarchical monotone behavior, and can simplify the proof of liveness significantly. These predicates, though very intuitive for manual reasoning, are hard to mine automatically. We add these predicates as additional signals in the design, and by allowing our extended $k$-LIVENESS algorithm to discover hierarchical monotonicity among these predicates, we achieve a speed-up in the final proof of liveness. See Section V for details.

As an additional (minor) contribution, we demonstrate how we can use the core idea of $k$-LIVENESS for proving response property in a more direct, and intuitive way avoiding the conventional approach of constructing Büchi automata. This leads to a quick, and simple implementation. See Section VI for details.

The paper is organized in the following sections: we overview the background concepts in the next section. In three of its subsections, we discuss our formal model of communication fabrics called xMAS, the And-Inverter graph representation for sequential circuits, and the formal specification of response property. Section III is a quick recapitulation of the basic $k$-LIVENESS algorithm. We present the notion of disjunctive stabilizing constraint, and the corresponding mining algorithm in Section IV. Communication fabric specific design hints are discussed in Section V. Section VI presents how we leverage the core idea of $k$-LIVENESS to create a model checker for the response property. Section VII
presents our experimental results. Section VIII discusses related work, and Section IX concludes the paper.

II. TERMINOLOGIES AND BACKGROUND FORMALISMS

A. Formal Model

Communication fabrics are typically constructed using finite FIFO buffers, sources and sinks of flits, function blocks acting on flits, decision primitives like switches and arbiters, and synchronization primitives like forks and joins. While these structural components can have various implementations, we follow a particular formal model called xMAS [9]. xMAS provides a clear logical specification of all these components. Fabrics designed in this formalism become synchronous Boolean circuits triggered by a single global clock. We refer to [9] for details of the formal specification of the components. Here we provide only an informal summary of the semantics of the components, with their symbols shown in Figure 1. A fork takes a flit from its input, and produces two new flits in two outputs, while a join does the dual operation. An arbiter arbitrates between two of its inputs, breaking ties as per its own arbitration policy. A switch sends the input flit to either of its outputs based on the type of data in the flit. Sources and sinks are producers and consumers of flits respectively. All sources and sinks considered in this paper are non-deterministic. Function units are computation blocks that may transform the control and/or data part of the flit. FIFO buffers are standard first-in-first-out buffers that can store finite number of flits. A communication fabric is made of these components by connecting them with channels. Each channel consists of three kinds of signals, viz. req (for request), gnt (for grant), and data. A channel has one xMAS component (called an initiator) on one side that initiates a transaction, and another component (called target) on the other side that receives the transaction. req and data signal go from initiator to target, and gnt goes from target to initiator. These signals are persistent, i.e. if a req is asserted by an initiator, it remains asserted until a corresponding gnt is asserted by the corresponding target. A fair understanding of xMAS semantics is assumed for the rest of the paper.

Fig. 1. xMAS symbols for structural components of communication fabrics

B. And-Inverter Graph

A Boolean graph is a directed acyclic graph (DAG) with nodes corresponding to logic gates, and directed edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states. A node \( v \) has zero or more fanins, i.e. nodes that are driving \( v \), and zero or more fanouts, i.e. nodes driven by \( v \). The primary inputs (PIs) are nodes without fanins in the current network. The primary outputs (POs) are a subset of nodes of the network. Any combinational Boolean function can be represented with AND gates, and NOT gates only. Any circuit can, therefore, be represented with AND gates, NOT gates, and D-flip-flops. The resulting graph structure is called And-Inverter graph (AIG). It is the standard data-structure being used in most of the recent academic, and industrial verification tools. We will use AIGs to represent the bit-blasted versions of communication fabrics. Hence our algorithms are developed, and experiments are performed around AIGs.

C. Formulation of Response Property

The mathematical formulation of the response property that we are interested in is presented in this section. Instead of introducing the formalism in an abstract set up, we choose to illustrate it around an example. For a more theoretical and generic discussion on response properties, see [17]. Consider the model shown in Figure 2. It is an implementation of a virtual channel in xMAS. Here a single physical channel (channel \( e \)) is shared between two sources (sources \( A_1 \) and \( A_2 \)). Virtual channel is a fundamental building block of on-chip communication networks which was invented to mitigate the head-of-line (HOL) blocking problem in wormhole switching. See texts like [13], [14] for details. Figure 2 shows how two sources \( A_1 \) and \( A_2 \) share the virtual channel \( e \) to transfer flits to their respective sinks (sink\(_1\) and sink\(_2\) respectively) while credit logic blocks CL1 and CL2 control flow of flits from source \( A_1 \) and \( A_2 \) respectively. We refer to the model of Figure 2 as \( \forall e \) in the subsequent sections.

As per xMAS formalism, each channel \( u \) consists of three types of signals, viz. request (u.req, 1-bit signal), grant (u.gnt, 1-bit signal), and data (u.data, bit-vector signal). In Figure 2, the sender at the source \( A_1 \) asserts the signal \( a_1.req \) when it wants to send a flit. When the network is ready to accept the flit from \( A_1 \), it will assert the signal \( a_1.gnt \). Therefore, the designer’s objective will be to prove that whenever the source \( A_1 \) makes a request to send a packet, it will be eventually granted. This will prove that \( A_1 \) will never be ‘blocked forever’. In LTL (Linear Temporal Logic), it can be written as \( G(a_1.req \Rightarrow F(a_1.gnt)) \). It is a well-known liveness property. While model checking this property, one needs to make some fairness assumptions on the sinks and (some of) the sources that they will never cease of work. Otherwise, one may come up with a rather uninteresting buggy scenario where some request from \( A_1 \) is never granted simply because one or more sinks stopped to drain flits, or credit source in CL1 stopped to supply credits. This contradicts a basic assumption that the sinks and the sources model the (non-deterministic) environment that may not react immediately, but will not cease to react either. For the virtual channel of Figure 2, these necessary fairness assumptions may be written in LTL as \( GF(m_1.gnt) \), \( GF(m_2.gnt) \), \( GF(j_1.gnt) \), \( GF(j_2.gnt) \) for the sinks, and \( GF(i_1.req) \) and \( GF(i_2.req) \) for the credit sources. The overall proof objective \( \phi_{response} \)

\(^1\)flits are units of data transfer in communication fabrics, see [13] for details.

The formulation is uniform across the entire family of fabrics, and has no particular relation to the virtual channel design or implementation that we are considering here. For example, the same formulation applies to all other fabrics considered in this paper.
(or $\phi_r$ in short) is, therefore,

$$\phi_r \triangleq ((\textit{sink\_fair} \land \textit{source\_fair}) \Rightarrow \textit{response})$$

where

$$\textit{sink\_fair} \triangleq \textit{GF}(m_1,\textit{gnt}) \land \textit{GF}(m_2,\textit{gnt}) \land \textit{GF}(j_1,\textit{gnt}) \land \textit{GF}(j_2,\textit{gnt})$$

$$\textit{source\_fair} \triangleq \textit{GF}(i_1,\textit{req}) \land \textit{GF}(i_2,\textit{req})$$

$$\textit{response} \triangleq \textit{G}(a_1,\textit{req} \Rightarrow \textit{F}(a_1,\textit{gnt}))$$

A similar formulation applies to channel $A_2$ as well. Note that the above formulation treats the design almost as a blackbox, and does not refer to its internal signals or topology. Therefore, by careful selection of fairness constraints, the above generic formulation of response property $\phi_r$ can be easily applied to any other $\chi$MAS design.

### III. $k$-Liveness Algorithm

$k$-Liveness is a brand-new algorithm for liveness verification, proposed by Claessen and Sörensson in [10]. It has outperformed other state-of-the-art liveness verification algorithms in the recent-most hardware model checking competition [2]. $k$-Liveness is designed to prove an LTL property of the form $\textit{FG}p$ on a finite state system represented as a sequential circuit. Since any $\omega$-regular property can be transformed into the form of $\textit{FG}p$ using generalized Büchi automata construction [21], in principle $k$-Liveness serves as a proof tool for the whole family of $\omega$-regular properties. The algorithm works in two phases, namely, a pre-processing phase, and an iterative proof phase. The iterative proof phase is the core proof engine of the algorithm, and it is both sound and complete even without the pre-processing phase. However, it is demonstrated that the pre-processing phase significantly simplifies, and expedites the iterative proof phase, and hence designated as an indispensable counter-part of the proof phase. Below, we describe the iterative proof phase first, and then the pre-processing phase.

#### A. Iterative Proof Phase

Suppose property $\textit{FG}p$ indeed holds on a design for some design signal $p$. It means that signal $p$ may toggle initially, but after some (finite) time, it will assume logic value 1, and remain 1 forever.

$$\textit{FG}p : \quad p$$

Fig. 3. A typical waveform for a signal $p$ that satisfies property $\textit{FG}p$

In other words, if $p$ assumes logic value 0 for at most $k_{\text{max}}$ number of times, then $k_{\text{max}}$ is a finite integer. In the iterative proof phase, $k$-Liveness searches for this bound $k_{\text{max}}$ iterating over all integers starting from 0. At the $i$-th iteration, for $i \geq 0$, it verifies if signal $p$ can assume logic value 0 for no more than $i$ number of times. It is indeed a safety verification obligation. If a safety verification tool proves the obligation, then $k_{\text{max}} = i$, and $\textit{FG}p$ is proved on the design. Otherwise $k$-Liveness moves to the next iteration, and checks if $p$ can assume logic value 0 for no more than $i+1$ number of times. If $k_{\text{max}}$ is indeed finite, the iteration will eventually terminate.

Clearly, this core idea of $k$-Liveness is engineered toward finding a proof of satisfaction of $\textit{FG}p$, and as it is, it does not terminate if the design admits a counterexample to $\textit{FG}p$. In order to make the algorithm complete, it has been integrated with a BMC phase to check if any counterexample of length $i$ can be found before moving to the $(i+1)$-th iteration. In the current work we too are interested in proofs, and not much in counterexamples. We, therefore, skip the details of integrating BMC with $k$-Liveness, and refer to [10] for further details.

#### B. Pre-processing Phase

In the pre-processing phase, $k$-Liveness mines a particular kind of design invariants, called stabilizing constraints. These invariants simplify proof obligations in the subsequent iterative proof phase. The rules for identification of stabilizing constraints, as proposed in [10], are the following: when $a$ is a design signal (i.e. either a gate output, or a register output of AIG $S$),

(R1) if $S \models \textit{G}(a \Rightarrow \textit{X}(a))$, then $\textit{FG}(a \Rightarrow \textit{X}(a))$ is a stabilizing constraint.

(R2) suppose $S \models \textit{FG}(a \Rightarrow \textit{X}(a))$, and additionally $S \models \textit{FG}(a \Rightarrow p)$ holds where $S \models \textit{FG}p$ is our original liveness proof obligation, then $\textit{FG} \neg a$ is a stabilizing constraint.

(R3) dual to rule (R2), suppose $S \models \textit{FG}(a \Rightarrow \textit{X}(a))$, and additionally $S \models \textit{FG}(\neg a \Rightarrow p)$, then $\textit{FG}a$ is a stabilizing constraint.

Claessen and Sörensson proposed an induction based approach to quickly discover an under-approximation of the set of all stabilizing constraints present in a design. If $M = \{m_1, m_2, \ldots, m_n\}$ be the set of stabilizing constraints identified by some algorithm based on the above three rules, $k$-Liveness is invoked on an AIG $S$ to prove the constrained property $\textit{FG}(\land_{i=1}^n m_i \Rightarrow p)$ (see [10] for justification).

### IV. Disjunctive Stabilizing Constraints

Our main contribution in this work is an augmentation of the notion of stabilizing constraints. It is done by generalizing the domain of its candidates. As proposed in rule (R1) above, $k$-Liveness scans over all signals present in the design, and checks which of them satisfies (R1). In particular, implementation of $k$-Liveness tests all gate outputs and register outputs of the AIG representing the design. Now, we observe that some predicates (i.e. Boolean functions) defined over some subset of design signals may also satisfy (R1), and may simplify the subsequent proof obligations significantly. These predicates can range over simple functions like conjunctions, implications, or complex, not-so-intuitive condition computations. Unfortunately, if these predicates were not present in the AIG as outputs of some gates (or registers), the native pre-processing step of $k$-Liveness as implemented in [10] would not be able to find them, and leverage them in the proof. With this insight, we propose to enrich the domain of candidate signals for the test of rule (R1) such that the tester would not only investigate signals that are already present in the AIG, additionally investigate some Boolean combination of existing signals as well.

As a general problem, the question of considering unknown Boolean combination of design signals for their stabilizing behavior is very hard to answer. Here we neither know the definitions of the target Boolean functions, nor we know their supports, nor even the support sizes. However, based on our application domain, i.e. response verification of communication fabrics, we can envision a particular class of Boolean functions that can exhibit stabilizing behavior, and can potentially simplify the proof obligation. This class has the following characterization: suppose $a_1$ is a design signal that satisfies $S \models \textit{G}(a_1 \Rightarrow \textit{X}(a_1))$. For such an $a_1$, we would like to find another signal $a_2$ such that $S \not\models \textit{G}(a_2 \Rightarrow \textit{X}(a_2))$, but $S \models \textit{G}((a_1 \lor a_2) \Rightarrow \textit{X}(a_1 \lor a_2))$. We call signals that can qualify as $a_1$ ‘level-1 stabilizing constraints’, and signals that can qualify as $a_2$ ‘level-2 stabilizing constraints w.r.t. $a_1$’. In this way, we can generalize this notion upto level-$\tau$ stabilizing constraint for some $\tau \geq 1$. Formally, a level-$\tau$ stabilizing constraint can be defined with the following recursive definition:

1. A signal $a$ is a level-1 stabilizing constraint iff $S \models \textit{G}(a_1 \Rightarrow \textit{X}(a_1))$

2. A signal $a_\tau$ is a level-$\tau$ stabilizing constraint, for $\tau > 1$, iff $S \models \textit{G}((a_1 \lor a_2 \lor \ldots \lor a_\tau) \Rightarrow \textit{X}(a_1 \lor a_2 \lor \ldots \lor a_\tau))$, but $a_\tau$ is not a level-$(\tau - 1)$ stabilizing constraint.

In the original $k$-Liveness algorithm, only level-1 stabilizing constraints were considered in the pre-processing step. In our pre-processor, our objective is to consider upto level-$\tau$ for some $\tau > 1$. We set $\tau$ as
Parameter whose value would depend on the available computational resource. Algorithm 1 presents our procedure of finding stabilizing constraints upto level-τ.

### Algorithm 1: Identification of stabilizing constraints

1. $C$ : set of candidate signals;
2. $L_0 \leftarrow \emptyset$;
3. $\tau \leftarrow 1$;
4. repeat
   5. $\tau \leftarrow \tau + 1$;
   6. $M_\tau \leftarrow \bigcup_{\tau=0}^{\tau-1}\{x|x$ appears in any one of the disjuncts in $L_\tau\}$;
   7. $C \leftarrow C \setminus M_\tau$;
   8. forall the $a \in C$ do
      9. forall the $l \in L_{\tau-1}$ do
         10. if $S \models G[(l \lor a) \Rightarrow X(l \lor a)]$ then
             11. $L_\tau \leftarrow L_\tau \cup (l, a)$;
   12. until ($\tau > \text{Threshold}$) $\lor (C = \emptyset) \lor (L_\tau = \emptyset)$;

Note that for a level-$\tau$ stabilizing constraint $a_\tau$, signal $a(\tau) = a_1 \lor a_2 \lor \ldots \lor a_\tau$ becomes a level-1 stabilizing constraint, with a difference that a $a(\tau)$ may not be present in the original AIG. Though signals $a_1 ... a_\tau$ are present. Therefore, our augmented domain of candidate signals is essentially the disjunctive domain over existing signals with parameterized support size. We specialize this disjunctive domain further by imposing the (recursive) restriction that $a_\tau$ cannot be a level-($\tau-1$) stabilizing constraint. This restriction stems from an observation that in communication fabrics, for a level-1 stabilizing constraint $a_1$, there might exists a (level-2 stabilizing) signal $a_2$, such that $\neg a_1 \Rightarrow a_2$ is a level-1 stabilizing signal. Similarly, there might exist a (level-3 stabilizing) signal $a_3$ such that $\neg a_1 \Rightarrow (\neg a_2 \Rightarrow a_3)$ is again a level-1 stabilizing signal. Note that the general form $\neg a_1 \Rightarrow (\neg a_2 \Rightarrow \ldots (\neg a_{r-1} \Rightarrow a_r) \ldots)$ is equivalent to $a_1 \lor a_2 \lor \ldots \lor a_r$. Apart from capturing this communication fabric specific requirement, this restriction over the general disjunctive domain prevents from the potential blow-up associated with considering all possible disjunctions for some support size.

### V. xMAS-specific predicates

For xMAS communication fabrics, we observe that the information whether some FIFO buffer is empty or not, or whether some buffer is full or not, has a very natural connection to how the fabric eventually issues grant to a request. In a bit-blasted fabric, however, this information are not readily available. A specialized algorithm is required that would identify appropriate register variables (associated with a particular FIFO buffer), and then synthesize appropriate Boolean function corresponding to the information whether a buffer is empty or not. Without any prior hint, it would be challenging for a general purpose liveness algorithm, like $k$-LIVENESS, to find such specialized information efficiently. Based on this observation, we introduce this information as explicit predicate signals in the design. In particular, for each buffer $B$ in the fabric, we introduce predicates $is\ _empty(B)$, and $is\ _full(B)$ in the AIG. The predicates have the following obvious definitions: $is\ _empty(B) = 1$ iff buffer $B$ is empty; $is\ _full(B) = 1$ iff buffer $B$ is full. During our experiments, before we invoke Algorithm 1 to discover stabilizing constraints for a design, we initialize the set of candidate signals $C$ as $\{is\ _empty(B), \neg is\ _empty(B), is\ _full(B), \neg is\ _full(B)||B$ is a buffer in the fabric}.

### VI. k-LIVENESS algorithm for response verification

As an $\omega$-regular property, response property $\phi_{r}$ can also be translated into the form of $\text{FG}p_{0}$ using generalized B"uchi automata construction. But we observe that we can use the core idea of $k$-LIVENESS to prove $\phi_{r}$ in a more direct way that avoids such translation. We illustrate the idea through the concrete example of $\forall C$ as follows:

If $\phi_{r}$ is to hold on $\forall C$, i.e. if every $a_i, req$ is eventually followed by a $a_i, gnt$, the interval for which $\forall C$ waits for $a_i, gnt$ once $a_i, req$ is asserted (called the pending interval) can never be infinite. As the fairness signals are assumed to be asserted infinitely often, the number of times the fairness signals are asserted during any pending interval can never be infinite either. So, $k$-LIVENESS algorithm can be used to prove that the number of times fairness signals are asserted during any pending interval is finite for $\forall C$.

A monitor that implements the above idea can be constructed as shown in Algorithm 2. This monitor is to be attached on $\forall C$, and $k$-LIVENESS is to be called to prove that signal $all\ _fairCount$ can assume logic value 0 only a finitely many times. Algorithm 2 is written in an imperative pseudocode with standard semantics where all assignments are to be evaluated parallelly at every clock. We adopt Verilog-style semantics for the data-types, namely, $wire$ represents combinational signals, and $reg$ represents sequential signals (i.e. registers). All $reg$ variables are initialized to logic 0, and for a $reg$ variable $r$, $next(r)$ computes its next state value.

### Algorithm 2: k-LIVENESS Monitor for Proving $\phi_{r}$ on $\forall C$

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

Algorithm 2 demonstrates how the core idea of $k$-LIVENESS can be used to verify $\phi_{r}$ in a direct way, without constructing a generalized B"uchi automata for the whole property $\phi_{r}$. However, this basic monitor of Algorithm 2 does not leverage the stabilizing constraints that we proposed in Section IV. Below, we show how Algorithm 3 modifies Algorithm 2 to include influence of the stabilizing constraints. In Algorithm 3, $M$ stands for the set of all stabilizing constraints mined in the pre-processing step. The roles of stabilizing constraints is captured in the variable arenaViolation.

#### A. Working principle, and correctness of the algorithms

In both Algorithm 2, and Algorithm 3, we use a primary input $oracle$ to model the non-deterministic choice for an arbitrary pending interval. It helps us trigger the monitor at an arbitrary time step when $a1,req$ is asserted, but $a1, gnt$ is not. Once this event occurs, register $oracleSaved$ remembers the event, and disables $oracle$ forever. This event also opens up the window of pending interval under examination. This window is closed in Algorithm 2 when a subsequent $a1,gnt$ arrives. Note that this construction works because there is no obligation of matching an $a1,gnt$ with a corresponding $a1,req$, and $a1,req$ has the persistence property [8]. Now, Algorithm 3 differs from Algorithm 2 in the way it closes the pending interval. When it opens up the pending interval, it takes a snap-shot of all signals of the form $a_i \lor \ldots \lor a_r$ in registers \{fair\}.
Algorithm 3: Arena-aware \(k\)-LIVENESS Monitor

\[
\text{Algorithm 3: Arena-aware } k\text{-LIVENESS Monitor}
\]

1. wire \(\text{pending, pendingInterval, allFair}\);
2. wire \(\text{fair}[6] = \{i_1, \text{req}, i_2, \text{req}, i_3, \text{gnt}, j_2, \text{gnt}, m_1, \text{gnt}, m_2, \text{gnt}\};
\]
3. reg \(f_1, f_2, \ldots, f_{|M|};
\]
4. reg \(\text{oracleSaved, gntSaved, fairFlop[6]}\);
5. forall the \(m \in M\) do
6. if \(\text{oracle} \land \neg\text{oracleSaved} \land \text{pending}\) then
7. \(\text{next}(f_m) := m;\)
8. else
9. \(\text{next}(f_m) := f_m;\)
10. \(\text{arenaViolation} := \bigvee_{m \in M} (f_m \neq m);\)
11. \(\text{pending} := a_1, \text{req} \land \neg a_3, \text{gnt};\)
12. \(\text{next(oracleSaved)} :=\)
13. \(\text{oracleSaved} \lor (\text{oracle} \land \neg\text{oracleSaved} \land \text{pending});\)
14. \(\text{next(gntSaved)} :=\)
15. \(\text{gntSaved} \lor (\text{oracle} \land (a_1, \text{gnt} \lor \text{arenaViolation}));\)
16. \(\text{pendingInterval} := \text{oracleSaved} \land \neg\text{gntSaved};\)
17. \(\text{allFair} := \bigwedge_{i=1}^{n} \text{fairFlop[i]};\)
18. for \(i \in \{1, 2, \ldots, 6\}\) do
19. if \(\text{allFair}\) then
20. \(\text{next(fairFlop[i])} := 0;\)
21. else
22. \(\text{next(fairFlop[i])} := \text{fairFlop[i]} \lor \text{fair[i]};\)
23. \(\text{allFairCount} := \neg(\text{pendingInterval} \land \text{allFair});\)

where signals \(a_i\) are level-\(i\) stabilizing constraints discovered in the pre-processing phase. Then through variable \(arenaViolation\), it keeps track of whether any one such signal has changed its value \((i.e.\) the check \(f_m \neq m)\). Since each such signal cannot flip its value within a potential counter-example to response property, a scenario under which it changes its value cannot be a candidate for violation to \(\phi_i\). If any such one signal has indeed changed its value, then the algorithm closes the pending interval (through \(a_1, \text{gnt} \lor \text{arenaViolation}\)). Note that Algorithm 3 achieves an early closure of the pending interval compared to Algorithm 2 due to the use of disjunctive stabilizing constraints. This helps in subsequent reduction in the number of iterations in the proof phase. It is straightforward to see that these constructions work for general situation of response verification as well.

VII. EXPERIMENTAL RESULTS

We use ABC [1] as our verification environment for both safety and liveness verification. We have developed our own Verilog implementation of XMAS library closely following the logical definitions given in [9], and used this library to implement various models of communication fabrics. We use our in-house tool VERIABC [16] to bit-blast these Verilog models. VERIABC uses VeriC, a commercial Verilog front-end analyzer, to read in the Verilog models. The benchmarks used are industrially relevant, and their designs are available in literature ([9], [15], [9]). They represent basic virtual channel (VC), virtual channel with channel buffer (VCB), virtual channel with ordering (VCO), and cascaded virtual channel with ordering (CVCO).

In [10], the authors compared their technique with liveness-to-safety (L2S) conversion [19]. They demonstrated that the latter is a close contender of \(k\)-LIVENESS algorithm. Liveness-to-safety transformation converts a liveness verification problem into an equi-satisfiable safety verification problem. It is a promising liveness verification technique currently adopted in industry to solve challenging verification problems [3]. For comparison purpose, we ran our benchmarks with our own implementation of liveness-to-safety algorithm, and results are shown in Table I. We used property directed reachability (PDR, a.k.a. IC3) [6] as the safety proof engine on the final single-output safety property instance. Verification run-times are shown in seconds in column ‘after L2S’. We used a time-out of 3600 seconds. Time taken by liveness-to-safety algorithm to solve these benchmarks in spite of their relatively small register counts demonstrate inherent complexity of the liveness problem we are addressing.

<table>
<thead>
<tr>
<th>Design</th>
<th>#PI</th>
<th>#flop</th>
<th>after L2S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual Channel (VC)</td>
<td>8</td>
<td>59</td>
<td>12.78</td>
</tr>
<tr>
<td>Virtual Channel with Buffer (VCB)</td>
<td>8</td>
<td>77</td>
<td>1515.20</td>
</tr>
<tr>
<td>Virtual Channel with Order (VCO)</td>
<td>9</td>
<td>104</td>
<td>126.44</td>
</tr>
<tr>
<td>Cascaded Virtual Channel with Order (CVCO)</td>
<td>12</td>
<td>182</td>
<td>&gt; 3600</td>
</tr>
</tbody>
</table>

TABLE I
LIVENESS PROPERTY WITHOUT ANY SAFETY INVARIANT

The experiment reported in Table I was performed on a laptop (referred to as ‘CPU1’ in the subsequent discussion) with a dual-core Intel Celeron 1.20 GHz processor, and 2GB memory. We compare performance of L2S with the \(k\)-LIVENESS algorithm in Table II. Performance of L2S from column 4 of Table I is replicated in column 2 of Table II. \(kcs\) is the command that invokes our implementation of \(k\)-LIVENESS in ABC. We performed similar experiments on a more powerful computer referred to as CPU2, and the run-times on CPU1 and CPU2 are reported in columns 3, and 4 respectively. CPU2 is a cluster of 4 Intel(R) Xeon(R) E5-2670 processors each with eight 2.6GHz cores, each with 20MB cache, and a total of 132GB RAM. The experiments of columns 3, and 4 were carried out with a switch \((-c\) of command \(kcs\)) that allows users to supply model specific invariants manually. We supplied a set of known design invariants [8] for our models. Interestingly, design CVCO could not be solved even with the user supplied invariants within the given time-out of 3600 seconds on CPU1. CPU2 solved it with its enhanced computing power, but took long time. Column 5 shows the values of \(k\) \((i.e.\) number of iterations in the proof phase) which \(k\)-LIVENESS took to prove individual designs when it converged on them. Apparently, the switch \(kcs\) - \(c\) is an interesting addition to the standard \(k\)-LIVENESS algorithm since it is absolutely important to have provision for user supplied invariants as they often help to get the proof faster. It should be mentioned that run-time of L2S in column 2 indeed took the user-supplied invariants into account, but still the process was much slower.

<table>
<thead>
<tr>
<th>Design</th>
<th>(kcs) - (c)</th>
<th>CPU1</th>
<th>CPU2</th>
<th>(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC</td>
<td>12.78</td>
<td>0.35</td>
<td>0.12</td>
<td>2</td>
</tr>
<tr>
<td>VCB</td>
<td>1515.20</td>
<td>8.55</td>
<td>2.54</td>
<td>2</td>
</tr>
<tr>
<td>VCO</td>
<td>126.44</td>
<td>19.08</td>
<td>6.32</td>
<td>4</td>
</tr>
<tr>
<td>CVCO</td>
<td>(&gt; 3600)</td>
<td>(&gt; 3600)</td>
<td>957.77</td>
<td>6</td>
</tr>
</tbody>
</table>

TABLE II
RESPONSE VERIFICATION WITH SAFETY INVARIANTS

Table III, and Table IV present run-times for the same verification problems, but now using disjunctive stabilizing constraints. Columns marked with PPT, and PT represent pre-processing time, and proof time respectively. Columns marked with \(k\), and \#c represent iterations performed by \(k\)-LIVENESS in order to get the proof, and the total number of stabilizing constraints generated in the pre-processing phase respectively. Times in columns PPT, and PT are in seconds. One benefit of using disjunctive stabilizing constraints is reflected in column \(k\) as this number is now reduced compared to the corresponding numbers in Table II. This also reduces run-time (column PT) of the proof step, sometimes quite significantly. For example, the total run-time for solving CVCO with disjunctive constraints is 209.98 sec. + 522.25 sec. = 732.23 sec, which is much less than 957.77 sec required to solve the same benchmark without disjunctive constraints. As the use of disjunctive
stabilizing constraints achieves the proof with smaller number of $k$, it puts less stress on the safety verification engine, and increases chance of convergence of the final proof for challenging benchmarks. For proving all safety obligations, we used property directed reachability (PDR) engine of ABC. In the pre-processing phase, we discharged a number of verification obligations. We used PDR to prove these objectives one after another sequentially. We used only level-1, and level-2 stabilizing constraints in our experiments, and all experiments were repeated on CPU1, and CPU2. Clearly, the over-all run-time decreased remarkably with increase in computing power. But the pre-processing time was indeed high for every experiments. We devised another set of experiments reported in Table IV where we have reduced this pre-processing time significantly. It is described as follows:

<table>
<thead>
<tr>
<th>Design</th>
<th>CPU2</th>
<th>k</th>
<th>#c</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC</td>
<td>1.96</td>
<td>0.11</td>
<td>1</td>
</tr>
<tr>
<td>VCB</td>
<td>77.29</td>
<td>1.73</td>
<td>1</td>
</tr>
<tr>
<td>VCO</td>
<td>7.87</td>
<td>5.06</td>
<td>2</td>
</tr>
<tr>
<td>CVCO</td>
<td>209.98</td>
<td>522.25</td>
<td>3</td>
</tr>
</tbody>
</table>

Table III PERFORMANCE OF $k$-LIVENESS WITH arenaViolation

Liveness and Multi-output Safety: The stabilizing constraints are mined in the last experiment using PDR sequentially. If we suspect a signal $a$ to be stabilizing, we create a new signal corresponding to $a \Rightarrow Xa$. For a set of suspect signals, we create a set of such property signals, and discharge PDR to check them one after another. This sequentiality resulted in longer time for the pre-processing step. Now, the AIG with all the newly created property outputs can be seen as a circuit with multiple property outputs, and a proof engine specialized for solving multiple property outputs simultaneously can be used. Solving multiple properties simultaneously can offer certain advantage in run-time as we can learn useful information about the state-space for solving one property, and leverage the learned information to solve another property. Specialized solvers are being designed that may use specialized techniques for solving multi-property problems. The hardware model checking competition started a new track last year where solvers designed for this particular purpose can compete with each other. ABC has such an engine, again based on PDR, which can handle multiple properties together (available as command ‘pdr -a’). We used this engine to solve our suspect stabilizing properties in one shot, and the results were very promising as shown in Table IV. In every corresponding run (w.r.t Table III), we found that the pre-processing time reduced dramatically. Use of the same multi-property engine for solving the iterative proof phase also resulted in run-time improvement for most of the cases.

<table>
<thead>
<tr>
<th>Design</th>
<th>CPU2</th>
<th>k</th>
<th>#c</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC</td>
<td>0.43</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>VCB</td>
<td>6.19</td>
<td>1.29</td>
<td></td>
</tr>
<tr>
<td>VCO</td>
<td>1.43</td>
<td>2.09</td>
<td></td>
</tr>
<tr>
<td>CVCO</td>
<td>30.87</td>
<td>96.71</td>
<td></td>
</tr>
</tbody>
</table>

Table IV PERFORMANCE OF $k$-LIVENESS WITH arenaViolation

VIII. RELATED WORK

Ranking-oriented proof of liveness is certainly not a new research topic. The underlying core mathematical notions, viz. well-founded induction, and ranking function, are fundamental concepts in discrete mathematics. Based on these ideas, Zohar Manna and Amir Pnueli laid the foundation of ranking-oriented proof of liveness of general reactive systems in the 1980s [17]. In recent years, research interests in this area have been renewed, particularly in the context of termination analysis of software ([12], [11], [4], [18]). To the best of our knowledge, however, no prior work has addressed the problem of ranking-oriented proof of liveness for communication fabrics.

Among the recent works on liveness verification of communication fabrics, most closely related to our problem are [15], and [20]. Both of these two are based on XMAS formalism. In [15], Gotmanov et. al proposed an efficient technique for proving deadlock freedom of communication fabrics by composing sets of sufficient conditions for deadlock freedom of each individual XMAS component. This system-wide condition is checked with a SAT solver for unsatisfiability. Our algorithm also addresses the same problem, with a focus on verification of response property. In [20], Schmaltz et al. proposed algorithms based on graph analysis that detects deadlock freedom for network-on-chips represented using XMAS formalism. Their algorithm does not address the problem of bit-level verification, rather it certifies designs at the micro-architecture level.

Research on general algorithms for bit-level liveness verification has gained remarkable momentum in recent times. In hardware model checking world, new algorithms and tools are being developed. Biere et al. showed how algorithms for safety verification can be re-used for liveness verification through their liveness-to-safety conversion [5], [19]. Bradley et. al. proposed FAIR [7], a scalable, incremental algorithm for liveness verification based on their remarkably successful algorithm IC3 [6] for safety verification. $k$-LIVENESS is a recent addition to the arsenal of scalable liveness algorithms. Both $k$-LIVENESS, and our methodology have strong conceptual connection with FAIR. The term (and the notion of) ‘arena’ used in Algorithm 3 is inspired by FAIR.

IX. CONCLUSION AND FUTURE WORK

We present a generalization of the pre-processing phase of the brand-new $k$-LIVENESS algorithm. $k$-LIVENESS has established itself as the state-of-the-art of bit-level liveness verification algorithm by outperforming existing algorithms in the liveness track of the recent hardware model checking competition. By generalizing one of its key steps, we improve its effectiveness further on challenging benchmarks. Our generalization, called disjunctive stabilizing constraints, is inspired by applications of liveness verification for communication fabrics. We have experimented on fabric designs of industrial relevance, and demonstrated effectiveness of our approach. We believe this is only a first step, and opens up a new avenue of research. Disjunctive stabilizing constraint is a general notion that could be effective outside communication fabric application. Developing a well-engineered implementation of mining algorithm for disjunctive constraints is our next step. It would be interesting to investigate which other application domain can also benefit from this extended pre-processing step of $k$-LIVENESS algorithm. In our implementation, we used PDR to filter out signals that are not stabilizing. An induction based approach could discover an adequate subset of the set of all stabilizing signals up to level-$\tau$ rather quickly. It needs further experimentation to decide which technique should be used for challenging benchmarks. As a theoretical question, it would be interesting to investigate formal connection of level-$\tau$ stabilizing constraints, with the classical notion of ranking functions. In our extension of $k$-LIVENESS algorithm, we only considered rule (R1). It would be interesting to determine the roles of rules (R2), and (R3) in the definition of level-$\tau$ stabilizing constraints.

X. ACKNOWLEDGEMENT

We acknowledge Alan Mishchenko for many helpful discussions on ABC. Jiang Long for his help on bit-blasting our models, and our anonymous reviewers for their suggestions of improvements over the initial draft of the paper. This work has been supported in part by SRC, NSA, and our industrial sponsors: Actel, Altera, Atrenta, Cadence, Calypto, IBM, Intel, Jasper, Magma, Oasys, Real Intent, Synopsys, Tabula, and Verific.
REFERENCES