Formal Deadlock Verification for Click Circuits

Freek Verbeek, Sebastiaan Joosten and Julien Schmaltz
School of Computer Science, Open University of The Netherlands
E-mail: {freek.verbeek,bas.joosten,julien.schmaltz}@ou.nl

Abstract—Scalable formal verification constitutes an important challenge for the design of complicated asynchronous circuits. Deadlock freedom is a property that is desired but hard to verify. It is an emergent property that has to be verified monolithically. We propose to use Click, an existing library of asynchronous primitives, for verification. We present the automatic extraction of abstract SAT/SMT instances from circuits consisting of Click primitives. A theory is proven that opens the possibility of applying existing deadlock verification techniques for synchronous communication fabrics to asynchronous circuits.

1 Introduction

Formal verification becomes more and more key in the design of complex hardware like real-time, control and hybrid systems. Its main appeal is the covering of *all* possible behaviors of some circuit – including obscure corner cases and unexpected phenomena – as opposed to, e.g., simulation-based methods. In contrast, formal verification applies powerful mathematics-based methods to prove desired properties over some system.

The focus of this paper is the verification of deadlock freedom. The difficulty with deadlock verification is that deadlock freedom is an *emerging* property. Establishing deadlock freedom of primitives in isolation does not provide any information on deadlock freedom of the entire system. A monolithic approach is mandatory. The current state-of-the-art in formal verification of asynchronous designs focuses on low-level descriptions of small and isolated circuits. Therefore, many approaches cannot be used for deadlock verification.

Recently, Peeters et al. introduced Click primitives [1]. Click primitives are low-level hardware design templates for quasi delay-insensitive elements such as storages, forks, joins and distributors. Connected in a pipelined fashion, these primitives restore a high level of abstraction during the design phase, even where a close link to realistic hardware is maintained.

The main observation that underlies this paper is that Click primitives may provide an appropriate level of abstraction for formal verification. On one hand they are closely related to realistic hardware. On the other hand, we will show that Click circuits are efficiently verifiable.

Our contribution is the automatic derivation of SAT/SMT instances from Click circuits. If the instance is infeasible, the circuit is deadlock-free. Our approach is sound, but incomplete. State-of-the-art SAT/SMT solvers are able to deal with huge instances, ensuring the potential scalability of our approach. As a result we are able to monolithically and automatically verify Click circuits.

Consider, e.g., the network in Figure 1. The circuit is composed of six Click primitives. These primitives use handshakes a through f to establish mutual communication. The input injects packets which are duplicated by the fork. Two storages s_0 and s_1 buffer these packets. The join waits for two packets at its inputs and combines them into one packet, which is sent to the output.

Given this circuit, we automatically derive the following result:

$$\mathbf{Dead}(a) \iff ((s_0 \wedge \neg s_1) \vee (s_1 \wedge \neg s_0)) \wedge (s_0 = s_1)$$

In words, this formula states that there is a deadlock involving handshake a if and only if exactly one of the storages is full *and* the internal state of both storages is equal. The left hand side of the conjunct indicates that if, e.g., storage s_0 contains a packet but storage s_1 does not, a deadlock would occur. In this configuration, the fork will never be able to duplicate two packets, whereas the join will never be able to combine two packets. The right hand side indicates that *invariably* both storages will either both be empty or both be full. The formula is not satisfiable, i.e., there is no assignment of values to variables that makes the formula true. Consequently, there is no deadlock.

2 Click: primitives for asynchronous networks

A quasi delay-insensitive circuit is modelled as a reactive system communicating with an environment [2]. Communications occur by sending and receiving signals through wires. No assumptions are made on timing. If two signals are sent through two wires, no information is available to the circuit on when or in which order these signals were sent or received. A quasi delay-insensitive circuit behaves correctly under minimal timing assumptions on the environment [3].

Click primitives are delay-insensitive circuits such as joins, arbiters, or distributors [1]. All communications with the environment are based on two-phase handshake protocols. For each handshake with primitive

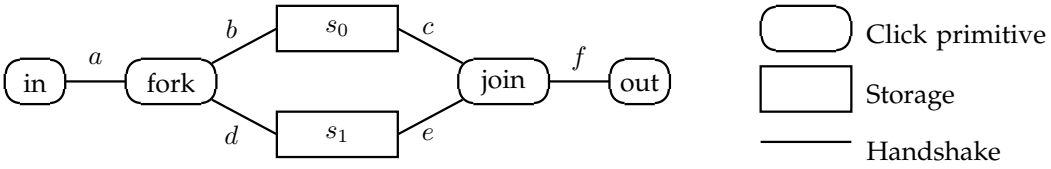

Fig. 1. Click circuit

p, two wires $p_{\mathtt{R}}$ and $p_{\mathtt{A}}$ are introduced for requests and acknowledgments. Input handshakes are passive, i.e., they wait for requests and send acknowledgments. Output handshakes are active. A main characteristic of Click primitives is that their internal state is updated at precisely defined moments, very similar to synchronous primitives.

Running Example, Part 1: Figure 2 shows the Click implementation of a join. The join waits for data from its two inputs a and b before forwarding data to output c. The register stores the internal state. The combinatorial logic captures the exact moment at which the state is updated and data can be forwarded, i.e., it is the *trigger condition*. In the initial state, signals $a_{\mathtt{R}}$, $b_{\mathtt{R}}$ and $c_{\mathtt{A}}$ are equal, which does not trigger a pulse on wire g. A request by either a or b occurs by a change on their respective request wires. When both $a_{\mathtt{R}}$ and $b_{\mathtt{R}}$ change, the trigger condition is satisfied. A pulse is sent to the register, triggering a change on wire $c_{\mathtt{R}}$. This invalidates the trigger condition, as $c_{\mathtt{R}}$ is now unequal to $c_{\mathtt{A}}$. The reception of an acknowledge on $c_{\mathtt{A}}$, i.e., a change on wire $c_{\mathtt{A}}$, is required to satisfy the third conjunct of the triggering condition. This reception returns the join to its initial state, with all signals inverted.

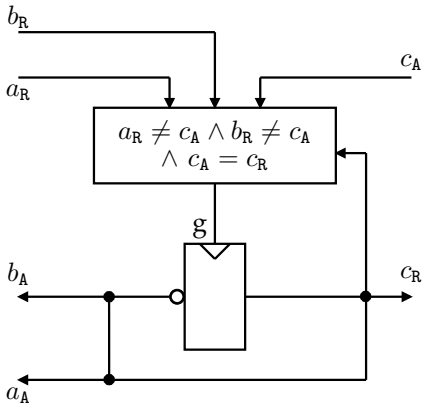

Fig. 2. Click implementation of a Join [1]

2.1 Formalization of Click Primitives

We represent Click primitives using the eXtended Delay Insensitive (XDI) specification [4], [5]. In this paper, XDI specifications are represented using automata. We introduce the parts of the XDI formalism relevant to this paper, using the join as running example (see Figure 3). An XDI specification consists of sets $W_{\mathtt{I}}$ and $W_{\mathtt{O}}$ of in- and output wires, a set of states S, and a set of transitions $\rightarrow$, each of which will be detailed.

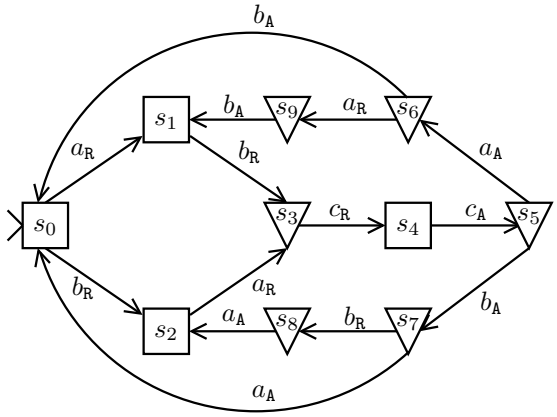

Fig. 3. XDI state graph of a join

Each Click primitive may be connected to several other primitives and may use several handshakes for this. We will use H to denote the set of handshakes. We allow the possibility that a request for handshake h is accompanied by data d. In this case, the handshake will be denoted with h^d.

Each handshake is implemented by two wires $w_{\mathtt{R}}$ and $w_{\mathtt{A}}$ for requests and acknowledgments. We denote the complete set of wires with W. If primitive X_T has request wire $w_{\mathtt{R}} \in W_{\mathtt{I}}$, it is a *target* of handshake w. Similarly, primitive X_I is an *initiator* of handshake w if it has a request wire $w_{\mathtt{R}} \in W_{\mathtt{O}}$.

For the join, the set of handshakes H is $\{a, b, c\}$. The set of input wires $W_{\mathtt{I}}$ is $\{a_{\mathtt{R}}, b_{\mathtt{R}}, c_{\mathtt{A}}\}$ and the set of output wires $W_{\mathtt{O}}$ is $\{a_{\mathtt{A}}, b_{\mathtt{A}}, c_{\mathtt{R}}\}$. The possibility of transmitting data with requests is not needed.

In contrast to the full XDI specification, which provides five different types of states, our presentation allows only two types of states: *indifferent* states (denoted with $\Box$) and *transient* states (denoted with ∇). An indifferent state poses no progress obligation on either the circuit or its environment. A transient state requires progress of the circuit, i.e., the primitive eventually has to proceed to a next state. Function $\tau : S \mapsto \{\Box, \nabla\}$ returns the type of the given state. Predicate $\mathrm{init} : S \mapsto \mathbb{B}$ returns true if and only if the given state is the initial state.

For the join, state s_0 is the initial state, i.e, $\mathrm{init}(s_0)$ returns true. As this state requires an input from the environment on wires $a_{\mathtt{R}}$ and $b_{\mathtt{R}}$, there are no progress

obligations and $\tau(s_0) = \square$. In state s_3, requests have been received from both a and b. The circuit has to send a request to its output c. Consequently, $\tau(s_3) = \nabla$.

The set of transitions $\rightarrow \subseteq S \times W \times S$ is labelled with wires. A transition $s_0 \xrightarrow{w} s_1$ occurs if and only if a communication occurs on wire w, i.e., when the wire changes value from low to high or from high to low. A transition is an input transition if and only if its label is an input. For details on rules on which transitions are allowed and required in XDI specifications, we refer to papers on the XDI formalism (e.g., [4]).

For the join, Figure 3 shows the set of transitions.

Definition 1: An XDI specification X is a state machine defined by the following parameters:

$$X \quad \stackrel{\text{def}}{=} \langle H, W_{\mathtt{I}}, W_{\mathtt{O}}, S, \tau, \text{init}, \rightarrow \rangle.$$

We will use a dot to access the elements of a tuple, e.g., $X.W_{\mathtt{I}}$ denotes the set of input wires of XDI specification X. If however the XDI specification is clear from the context, we will omit it. A complete *Click circuit*, denoted with C, consists of a set of mutually communicating Click primitives.

2.2 Execution Semantics

We use Linear Temporal Logic (LTL) to formalize properties over executions of Click circuits. LTL uses the $\mathbb{G}$(lobally) operator to express that some property is always true, and the $\mathbb{F}$(inally) operator which expresses that some property is eventually true [6]. We define an abstract notion of a *configuration* to capture the state of a Click circuit. The only relevant information for describing such a state is which packets are stored in which storages.

Definition 2: A *configuration*, notation σ, is an assignment of packets to storages.

An asynchronous execution of a Click circuit consists of transitions from configuration to configuration. We parameterize the LTL operators with a configuration σ, so that $\mathbb{G}_\sigma$ denotes "always from σ" and $\mathbb{F}_\sigma$ denotes "eventually from σ".

The execution semantics of an XDI state machine X are formalized relative to its environment. Since the environment consists of Click primitives, it is basically a large XDI state machine. The only information relevant to the analysis of primitive X, is whether its input wires are stable or not. If a wire w is stable, i.e., if its value is permanently unchanged, no transition labelled with w can be used. Therefore, the environment, i.e., the complete set of Click primitives constituting the circuit, is represented as a function which takes as parameter a wire in the circuit and returns a Boolean value representing whether the wire is stable.

Definition 3: An *environment*, notation ε, is a predicate over the wires W of the circuit.

$$\varepsilon \quad : \quad W \mapsto \mathbb{B}$$

The actual value of the environment depends on the state of the network. A configuration σ uniquely induces a value for the environment.

Definition 4: The *environment induced by configuration* σ, notation $\text{env}(\sigma)$, returns true if and only if eventually the given wire is stable in configuration σ.

$$\text{env}(\sigma)(w) \quad \stackrel{\text{def}}{=} \quad \mathbb{F}_\sigma(\mathbb{G}\, w \vee \mathbb{G}\, \neg w)$$

Running Example, Part 2: We consider an environment of the join in which wire $c_{\mathtt{A}}$ is stable. Any execution will strand in state s_4, i.e., waiting for the environment to acknowledge the receipt of data by output c after a request to c has been sent to fetch this data. Essentially, the join is dead because the environment permanently *blocks* on handshake c. In another environment wires $a_{\mathtt{R}}$ and $b_{\mathtt{R}}$ may be stable. The join will get stuck in its initial state s_0. It is waiting for the environment to send requests. Essentially, the join is dead because the environment is permanently *idle* on handshakes a and b. In total, the three input wires induce 2^3 different possible groups of environments of interest while analyzing the join, ranging from a live one (i.e., $\varepsilon(w)$ is false for $w \in \{a_{\mathtt{R}}, b_{\mathtt{R}}, c_{\mathtt{A}}\}$), to an environment where all three input wires of the join are stable.

We formally define execution traces with respect to an environment.

Definition 5: Given environment ε, an *execution* ρ is defined as a sequence of transitions labelled with unstable wires.

$$\rho = s_0 \xrightarrow{w_0} s_1 \xrightarrow{w_1} s_2 \xrightarrow{w_2} \ldots \quad \text{where} \quad \forall i \cdot \neg\varepsilon(w_i)$$

Basically, Definition 5 formalizes handshaking between input wires of X and the environment. An LTL can be built upon this definition of execution [7]. If, e.g., for primitive X in environment ε property p eventually holds, this is denoted:

$$X, \varepsilon \quad \models \quad \mathbb{F}\, p$$

3 Deadlock Formulae for Click

3.1 Deadlocks in Click circuits

A circuit has a deadlock if some primitive is infinitely trying to finalize a handshake with another Click primitive. For such a handshake a request has been sent, but permanently no acknowledge will be received. A structural deadlock is a configuration of the circuit which contains a handshake that eventually is permanently unfinished.

Definition 6: A configuration σ of a circuit C is a *structural deadlock*, notation $\omega(\sigma)$, if and only if it contains a handshake that eventually remains unfinished.

$$\omega(\sigma) \quad \stackrel{\text{def}}{=} \quad \exists h \cdot \mathbb{F}_\sigma((\mathbb{G}\, h_{\mathtt{A}} \vee \mathbb{G}\, \neg h_{\mathtt{A}}) \wedge h_{\mathtt{A}} \neq h_{\mathtt{R}})$$

In a structural deadlock there exists a handshake h whose acknowledgment wire $h_{\mathtt{A}}$ is eventually either permanently high or low. Additionally, the handshake must

be triggered, i.e., a request must have been transmitted. Note that we assume that the only cause of a change on wire $h_{\mathtt{R}}$ is when an acknowledgment arrives at $h_{\mathtt{A}}$. In other words, a Click primitive cannot drop his request. This assumption is called *persistency*.

A structural deadlock is not necessarily a *reachable* configuration, i.e., it might be a configuration that cannot be reached from the empty configuration. Let σ_0 denote the initial empty configuration. We define deadlock to be a reachable structural deadlock.

Definition 7: A Click circuit C has a *deadlock*, notation $\mathbf{\Omega}(C)$, if and only if there exists a reachable structural deadlock.

$$\mathbf{\Omega}(C) \quad \stackrel{\text{def}}{=} \quad \omega(\sigma_0)$$

A circuit is *deadlock-free* if and only if it has no deadlock.

3.2 Block- and Idle Formulae for Click

The central aspect of determining whether a primitive X causes a deadlock is to identify each non-transient state as *blocking* or *idling* handshake h. We define these labels in such a way that if primitive X is permanently stuck in a state labelled as "blocking h", handshake h is permanently blocked. Handshake h is permanently idle, if primitive X permanently remains in a state labelled "idling h".

Running Example, Part 3: For the join, we consider handshake a, which uses wires $a_{\mathtt{R}}$ and $a_{\mathtt{A}}$ to communicate with the join. States s_1, s_3, s_4, s_5, s_7, s_8 and s_9 are blocking this handshake. In these states, handshake a has sent a request to the join, which has not been acknowledged by the join yet. If the join is permanently stuck in these states, handshake a will permanently wait for an acknowledgment from the join. Handshake a is permanently blocked. The remaining states s_0, s_2, and s_6 are idling handshake a. In these states, the join waits for a request from handshake a. When it is permanently stuck in of these states, handshake a is failing to send this request. Handshake a is permanently idle.

Definition 8: Given an XDI specification X, predicate $\mathrm{blocking}$ is defined as the function satisfying the following constraint:

$$\begin{aligned}&\mathrm{blocking} : H \times S \mapsto \mathbb{B} \text{ such that}\\ &\quad \forall h^d \in H \cdot \forall x \in W, s_0 \xrightarrow{x} s_1 \cdot\\ &\quad \begin{cases} \neg\,\mathrm{blocking}(h^d, s_0) \wedge \mathrm{blocking}(h^d, s_1) & \text{if } x = h^d_{\mathtt{R}}\\ \mathrm{blocking}(h^d, s_0) \wedge \neg\,\mathrm{blocking}(h^d, s_1) & \text{if } x = h_{\mathtt{A}}\\ \mathrm{blocking}(h^d, s_0) \wedge \mathrm{blocking}(h^d, s_1) & \text{if otherwise}\end{cases}\end{aligned}$$

A state is idling handshake h if it is not blocking handshake h:

$$\mathrm{idling} \quad \stackrel{\text{def}}{=} \quad \neg\,\mathrm{blocking}$$

Predicate $\mathrm{blocking}$ takes as parameters a handshake h and a state s. It must change its value from false to true each time a transition occurs that is labelled with a request from handshake h. After such a transition, the primitive is in a state where it is dealing with the request. It is therefore blocking handshake h. After a transition labelled with an acknowledgment for handshake h, predicate $\mathrm{blocking}$ has to change value from true back to false. It is no longer blocking handshake h, as it has successfully dealt with the request.

Running Example, Part 4: The state machine of the join contains a transition $s_0 \xrightarrow{a_{\mathtt{R}}} s_1$. Definition 8 enforces $\mathrm{blocking}(a, s_0)$ to be false, whereas $\mathrm{blocking}(a, s_1)$ is true. This represents that state s_0 is idling handshake a, whereas state s_1 is blocking handshake a.

An important assumption on the Click primitives is that their XDI specification ensures that function $\mathrm{blocking}$ is unique over all non-transient states. If a certain state s can be reached from the initial state using a sequence of transitions with one transition labelled $h_{\mathtt{R}}$ and no transition labelled $h_{\mathtt{A}}$, Definition 8 enforces $\mathrm{blocking}(h, s)$ to be true. If this state s can also be reached with a sequence of transitions without $h_{\mathtt{R}}$ as label, Definition 8 enforces $\mathrm{blocking}(h, s)$ to be false. Such state graphs are not allowed. We will call a Click primitive for which function $\mathrm{blocking}$ is unique over all non-transient states *unambiguous*.

Whether a primitive can be stuck in a blocking- or idling state depends on the environment. Consider again the state machine of the join (see Figure 3). If the environment dictates that wire $c_{\mathtt{A}}$ is stable, any execution will end in state s_4. This state is blocking handshake a and therefore handshake a is permanently blocked. We say that a handshake h is permanently blocked if and only if in some environment a primitive will eventually get stuck in non-transient states labelled "blocking h".

Definition 9: Given environment ε, handshake h is *permanently blocked*, notation $\mathbf{Block}_\varepsilon(h)$, if and only if its target primitive X_T is eventually always in a blocking state. Similarly, handshake h is *permanently idle*, notation $\mathbf{Idle}_\varepsilon(h)$, if and only if its initiator X_I is eventually always in an idling state.

$$\begin{aligned}\mathbf{Block}_\varepsilon(h) &\stackrel{\text{def}}{=} X_T, \varepsilon \models \mathbb{F}\,\mathbb{G}\,\mathrm{blocking}(h) \vee \tau = \nabla\\ \mathbf{Idle}_\varepsilon(h) &\stackrel{\text{def}}{=} X_I, \varepsilon \models \mathbb{F}\,\mathbb{G}\,\mathrm{idling}(h) \vee \tau = \nabla\end{aligned}$$

To check for deadlock, we search for permanently blocked handshakes. A handshake that is permanently idle cannot cause a deadlock, as apparently no communications on the handshake wires occurs.

Definition 10: Given environment ε, handshake h is *dead*, notation $\mathbf{Dead}_\varepsilon(h)$, if and only if the handshake is permanently blocked and it is not permanently idle.

$$\mathbf{Dead}_\varepsilon(h) \quad \stackrel{\text{def}}{=} \quad \mathbf{Block}_\varepsilon(h) \wedge \neg\,\mathbf{Idle}_\varepsilon(h)$$

We prove that the existence of a structural deadlock is logically equivalent to the existence of a dead handshake.

Theorem 1: Configuration σ is a structural deadlock if and only if there exists a dead handshake in environment $\mathrm{env}(\sigma)$.

$$\omega(\sigma) \quad \Longleftrightarrow \quad \exists h \cdot \mathbf{Dead}_{\mathrm{env}(\sigma)}(h)$$

Proof:
($\Longrightarrow$) Assume handshake h is eventually stuck in a requesting state, i.e., $\mathbb{F}_\sigma((\mathbb{G}\, h_\mathtt{A} \vee \mathbb{G}\, \neg h_\mathtt{A}) \wedge h_\mathtt{A} \neq h_\mathtt{R})$. By Definition of $\mathbb{F}$, the circuit is eventually in a configuration σ' where $(\mathbb{G}_{\sigma'}\, h_\mathtt{A} \vee \mathbb{G}_{\sigma'}\, \neg h_\mathtt{A}) \wedge h_\mathtt{A} \neq h_\mathtt{R}$. We prove that handshake h is (1) not permanently idle, and (2) permanently blocked.
(1) We have to show that initiator primitive X_I is always eventually in a state labelled "not idling for h". We prove something stronger, namely:

$$X_I, \mathrm{env}(\sigma) \models \mathbb{F}\,\mathbb{G}\,\neg\,\mathrm{idling}(h)$$

In words, eventually primitive X_I is permanently stuck in blocking states. Let s_{X_I} be the state of primitive X_I that is reached by the execution yielding configuration σ'. We prove that in configuration σ' handshake h is blocked, i.e.,

$$X_I, \mathrm{env}(\sigma') \models \mathbb{G}\,\neg\,\mathrm{idling}(h)$$

In state s_{X_I}, a request has been sent since $h_\mathtt{A} \neq h_\mathtt{R}$. By Definition 8, state s_{X_I} is labelled blocking for h. Wire $h_\mathtt{A}$ is stable, i.e., $\mathbb{G}_{\sigma'}\, h_\mathtt{A} \vee \mathbb{G}_{\sigma'}\, \neg h_\mathtt{A}$. Wire $h_\mathtt{R}$ is stable since wire $h_\mathtt{A}$ is stable, and since we assume persistency on requests signals. By definition, environment $\mathrm{env}(\sigma')$ returns true for both wires. Therefore, any state reachable from s_{X_I} has been reached without transitions labelled $h_\mathtt{R}$ or $h_\mathtt{A}$. By Definition 8 any reachable state from s_{X_I} has the same label as state s_{X_I}, which is "blocking for h".

We have established that once primitive X_I reaches state s_{X_I} it is permanently not idling handshake h, i.e.,

$$X_I, \mathrm{env}(\sigma') \models \mathbb{F}\,\mathbb{G}\,\neg\,\mathrm{idling}(h)$$

It can be proven that this implies

$$X_I, \mathrm{env}(\sigma') \models \mathbb{G}\,\mathbb{F}\,\neg\,\mathrm{idling}(h)$$

This is logically equivalent to

$$\neg\,\mathbf{Idle}_{\mathrm{env}(\sigma')}(h)$$

since $\mathrm{env}(\sigma') = \mathrm{env}(\sigma)$ by Definition 4. (2) The proof that target primitive X_T is eventually always blocking handshake h is similar. Let s_{X_T} be the state reached by target primitive X_T after the execution yielding configuration σ'. Definition 8 ensures that state s_{X_T} is labelled "blocking for h". As wires $h_\mathtt{A}$ and $h_\mathtt{R}$ are stable, this label will not change. We have established

$$\mathbf{Block}_{\mathrm{env}(\sigma)}(h)$$

($\Longleftarrow$) Assume for some handshake h

$$X_T, \mathrm{env}(\sigma) \models \mathbb{F}\,\mathbb{G}\,\mathrm{blocking}(h)$$

Definition 8 ensures that the target of h is permanently in a state where a request has been received, but no acknowledge is sent back. This implies that eventually wire $h_\mathtt{A}$ is stable. Similarly,

$$X_I, \mathrm{env}(\sigma) \models \neg\,\mathbb{F}\,\mathbb{G}\,\mathrm{idling}(h)$$

implies that eventually a request will be triggered in $h_\mathtt{R}$. Consequently, configuration σ is a structural deadlock. □

3.3 Formulae per Primitive

We can now formulate block- and idle formulae for various Click primitives. We present formulae for some of the primitives presented by Peeters et al. [1]. These logical equations hold for *any* environment, i.e, variable ε is unbound.

The input of a join is permanently blocked if and only if either its output is permanently blocked or the other input is permanently idle.

$$\mathbf{Block}_\varepsilon(a) \iff \mathbf{Block}_\varepsilon(c) \vee \mathbf{Idle}_\varepsilon(b)$$

The output of a join is permanently idle if and only if one of its inputs is permanently idle.

$$\mathbf{Idle}_\varepsilon(c) \iff \mathbf{Idle}_\varepsilon(a) \vee \mathbf{Idle}_\varepsilon(b)$$

The *fork* is a primitive duplicating the packet arriving at input a to outputs b and c. The input of a fork is permanently blocked if and only if one of its outputs is permanently blocked.

$$\mathbf{Block}_\varepsilon(a) \iff \mathbf{Block}_\varepsilon(b) \vee \mathbf{Block}_\varepsilon(c)$$

The output of a fork is permanently idle if and only if one of either its input is permanently idle or the other output is permanently blocked.

$$\mathbf{Idle}_\varepsilon(b) \iff \mathbf{Idle}_\varepsilon(a) \vee \mathbf{Block}_\varepsilon(c)$$

The *merge* arbitrates between two inputs a and b and forwards packets to output c. We assume arbitration is *fair*, i.e., no starvation can occur. This assumption ensures that the input of a merge is permanently blocked if and only if its output is permanently blocked.

$$\mathbf{Block}_\varepsilon(a) \iff \mathbf{Block}_\varepsilon(c)$$

The output of a merge is permanently idle if and only if both its inputs are permanently idle.

$$\mathbf{Idle}_\varepsilon(c) \iff \mathbf{Idle}_\varepsilon(a) \wedge \mathbf{Idle}_\varepsilon(b)$$

A *storage* receives packets from input a, stores them and sends them forward to output b when possible. The storages are a special case, as blockage or idling of a storage depends on its internal state. For sake of presentation, the only information about the internal state of a storage we consider is whether it is full or not. In this case, the state of a storage can be represented with a Boolean variable. More complex state information, such as which data is buffered in the storage, can be represented using more complexly typed variables. Given storage $\underline{s}$ we introduce a variable s with intended meaning that variable s is true if and only if storage $\underline{s}$ is full. The input of a storage is permanently blocked if and only if the storage is full and its output is permanently blocked.

$$\mathbf{Block}_\varepsilon(a) \iff s \wedge \mathbf{Block}_\varepsilon(b)$$

The input of a storage is permanently blocked if and only if the queue is full and its output is permanently blocked.

$$\mathbf{Idle}_\varepsilon(b) \iff \neg s \wedge \mathbf{Idle}_\varepsilon(a)$$

The *distributor* (see Figure 4) is a Click primitive used for routing packets through a network. It uses handshake a on which the availability of data is communicated and four handshakes select^d ($00 \leq d \leq 11$). If $d = 00$, the incoming data is dropped. If $d = 01$, the packet is routed towards output b. Similarly $d = 10$ routes towards output c. If $d = 11$ data is copied to both outputs.

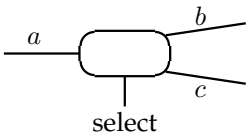

Fig. 4. Schematic overview of the distributor

Figure 5 shows the XDI state graph of the distributor. The set of input wires $W_\mathtt{I}$ is $\{a_\mathtt{R}, \text{select}^d_\mathtt{R}, b_\mathtt{A}, c_\mathtt{A}\}$. The remaining wires are output, i.e., $W_\mathtt{O} = \{a_\mathtt{A}, \text{select}_\mathtt{A}, b_\mathtt{R}, c_\mathtt{R}\}$. Handshake select uses data for requests.

Our theory does not apply to this distributor, as it is ambiguous. Predicate blocking can be defined in multiple ways, meaning that certain non-transient states are both blocking and idling. Consider state s_{12} and the sequence $s_0, s_5, s_6, s_8, s_9, s_{12}$ leading from the initial state to this state. The transition from s_0 to s_5 is labelled with a request on select^{11}. The remainder of the sequence contains no transition labelled with the corresponding acknowledgment select$_\mathtt{A}$. This indicates that when the distributor is in state s_{12}, it is still processing a request and is therefore blocking handshake select^{11}. Now consider sequence s_0, s_4, s_9, s_{12}. As this sequence does not contain any transition labelled select$^{11}_\mathtt{R}$, the distributor is not blocking handshake select^{11}. Since the value of $\text{blocking}(\text{select}11, s_{12})$ cannot be uniquely determined, the primitive is not unambiguous.

As a quick fix, we remove handshake select^{11} from the distributor. In contrast to the original distributor, the resulting primitive cannot be used as a fork to duplicate its incoming input. The resulting primitive is however unambiguous. The following block formula can be proven:

$$\mathbf{Block}_\varepsilon(a) \iff \begin{cases} \mathbf{Idle}_\varepsilon(\text{select}^{00}) \wedge \mathbf{Idle}_\varepsilon(\text{select}^{01}) \wedge \mathbf{Idle}_\varepsilon(\text{select}^{10}) \\ \neg\, \mathbf{Idle}_\varepsilon(\text{select}^{10}) \wedge \mathbf{Block}_\varepsilon(b) \\ \neg\, \mathbf{Idle}_\varepsilon(\text{select}^{01}) \wedge \mathbf{Block}_\varepsilon(c) \end{cases}$$

Mechanical Verification

Even though these formulae are simple and intuitive, their proofs are not obvious. The block- and idle formulae of the join, e.g., have to be proven correct in eight different types of environments (see Running Example, Part 2). Also, manually establishing that some primitive is unambiguous is tedious and error-prone.

We have applied the ACL2 theorem prover to establish correctness of these formulae[1] [8]. First, we have formalized the execution semantics of XDI state machines in the ACL2 logic. Secondly, we have implemented a function which returns either the predicate blocking of Definition 8 or an error value if the primitive is not unambiguous. Thirdly, we have made a parser for XDI state machines described in AND/IF_1.0 format[2]. Lastly, we have implemented an executable LTL for XDI state machines in the ACL2 logic and proven that this executable version is correct, e.g., $X, \varepsilon \models \mathbb{F}\, p$ returns true if and only if XDI state machine X in environment ε is eventually in a state where property p holds.

These four components allow us to automatically verify blocking- and idle formulae for Click primitives. We first parse an XDI state machine description. We ensure that it is unambiguous. Subsequently, all relevant environments are generated and for each environment the executable LTL decides whether the given formula is correct.

4 Unfolding the Formulae to SAT/SMT

Now that we have established block- and idle formulae for Click primitives, we can use them for deadlock detection. For each primitive, we unfold the block- and idle formulae. Since these formulae are necessary and sufficient conditions, they capture any behavior of the circuit. We explain the unfolding of block- and idle formulae by example.

Consider the asynchronous circuit in Figure 1. We assume that the in- and output of the circuit are fair, i.e., $\mathbf{Idle}_\varepsilon(a)$ and $\mathbf{Block}_\varepsilon(f)$ are both false. We are going to determine whether it is always possible to inject packets into this network, by unfolding $\mathbf{Dead}_\varepsilon(a)$. Note that the environment is still left uninterpreted, i.e., it is a free universally quantified variable.

$$\mathbf{Dead}_\varepsilon(a) \stackrel{\text{def}}{=} \neg\, \mathbf{Idle}_\varepsilon(a) \wedge \mathbf{Block}_\varepsilon(a)$$

Since the source is fair, only $\mathbf{Block}_\varepsilon(a)$ needs to be unfolded. The block formula of the fork is unfolded:

$$\mathbf{Block}_\varepsilon(a) \iff \mathbf{Block}_\varepsilon(b) \vee \mathbf{Block}_\varepsilon(d)$$

Consider the unfolding of the left disjunct. Storage s_0 must be full, in order for handshake b to be blocked:

$$\mathbf{Block}_\varepsilon(b) \iff s_0 \wedge \mathbf{Block}_\varepsilon(c)$$

Unfolding the block formula of the join gives:

$$\iff s_0 \wedge (\mathbf{Block}_\varepsilon(f) \vee \mathbf{Idle}_\varepsilon(e))$$

The output is fair. Storage s_1 must be empty for handshake e to be idle:

$$\iff s_0 \wedge \neg s_1 \wedge \mathbf{Idle}_\varepsilon(d)$$

1. Proof scripts are available online.
2. http://edis.win.tue.nl/and-if/index.html

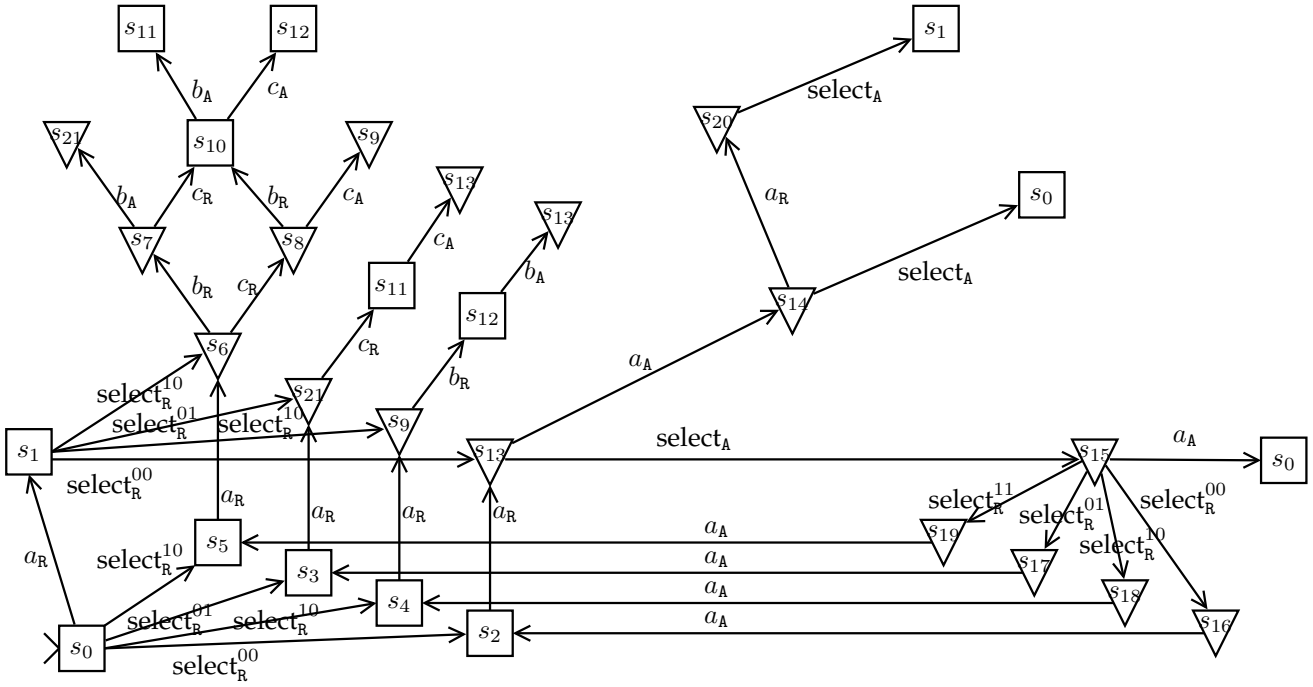

Fig. 5. XDI state graph of the distributor

Unfolding the idle formula of the fork gives:

$$\iff s_0 \wedge \neg s_1 \wedge (\mathbf{Idle}_\varepsilon(a) \vee \mathbf{Block}_\varepsilon(b))$$

The input is fair. Formula $\mathbf{Block}_\varepsilon(b)$ has already been unfolded. Thus this yields:

$$\iff s_0 \wedge \neg s_1$$

Similarly, $\mathbf{Block}_\varepsilon(d)$ is unfolded to:

$$\mathbf{Block}_\varepsilon(d) \iff s_1 \wedge \neg s_0$$

Which gives the result:

$$\mathbf{Dead}_\varepsilon(a) \iff (s_0 \wedge \neg s_1) \vee (s_1 \wedge \neg s_0)$$

The result is given to an off-the-shelf SAT solver. If the solver finds a solution, a *structural* deadlock exists. If not, the network is deadlock-free. By unfolding the deadlock formula $\mathbf{Dead}_\varepsilon(a)$, we have found two deadlock scenarios: one in which the upper storage is full and the lower one is empty, and the other way around.

Ultimately, we are not interested in structural deadlocks, but in actual reachable deadlocks. Additional invariants can be used to rule out unreachable configurations. Chatterjee et al. present a technique to automatically derive *flow invariants* for circuits similar to Click circuits [9]. Such a flow invariant relates configurations of different primitives in the circuit. For example, assume that in the initial configuration of the circuit in Figure 1 both storages are empty. An example of a flow invariant is:

$$s_0 = s_1$$

This invariant indicates that both storages are invariably both full or both empty. Adding the invariant to the SAT instance yields:

$$((s_0 \wedge \neg s_1) \vee (s_1 \wedge \neg s_0)) \wedge (s_0 = s_1)$$

This equation is infeasible. Consequently, the circuit is deadlock-free.

5 Related Work

Formal verification has been applied to great lengths in synchronous designs, e.g., [10], [11], [12]. Recently, formal methods have been applied intensively to the verification of asynchronous design. Günther et al. present a framework for verifying GALS-based architectures [13]. They manually create an abstract model, called a contract, of each synchronous component and model check an abstract GALS model obtained by combining these contracts. Their approach is however not scalable to sufficiently large designs and starts at a high level of abstraction. Yan et al. verify asynchronous circuits at a very low level of abstraction [14]. They use CoHo, a reachability analysis tool, to verify, e.g., that a full-buffer circuit behaves correctly. Ouchet et al. use simulations to verify robustness of various C-elements [15]. A comprehensive survey of formal verification on asynchronous circuits can be found elsewhere [16]. Most of these approaches suffer from the state space explosion problem. They consider small circuits in isolation. In contrast, we consider the integration of many asynchronous primitives and verify an emerging property, namely the absence of deadlocks.

Recently, Chatterjee et al. introduced xMAS (for: eXecutable Micro Architecture Specification) [17]. It consists of basic building primitives such as joins, forks, switches and arbiters, which interact in a *synchronous*

fashion. The use of block- and idle formulae to reduce the decision of deadlock freedom to an automatically verifiable set of equations has been applied before on these xMAS designs [18]. Using invariant generation and model checking, xMAS is suitable for *scalable* formal verification [19], [9]. However, the synchronous xMAS semantics differ greatly from the asynchronous behavior of Click circuits. Using the theory presented in this paper, these existing verification techniques for synchronous xMAS fabrics can possibly be used for asynchronous circuits.

6 Conclusion

This paper has presented a formal methods based approach for verifying deadlock freedom of asynchronous circuits consisting of Click primitives. From a Click circuit we automatically derive a SAT/SMT instance. Such an instance can be solved by off-the-shelf solvers. If the solver proves infeasibility of the SAT/SMT instance, the circuit is deadlock-free. Any solution corresponds to a structural deadlock. Using invariants we can rule out unreachable deadlocks.

Click primitives are very similar to xMAS primitives. Efficient deadlock detection tools for xMAS exist, which are based on block- and idle formulae equivalent to those presented in this paper. By proving such formulae, we effectively open up the possibility of using existing xMAS verification tools for asynchronous circuits.

Crucial for correctness of our approach are correct block- and idle formulae for each Click primitive. These formulae are often easy to understand, but their proofs are large and cumbersome. We have mechanically verified formulae for various Click primitives in the ACL2 theorem prover.

We have not yet applied our approach to interesting and relevant asynchronous examples. Experiments with non-trivial asynchronous circuits could help in assessing the practicality and scalability of our approach, and suggest further improvements.

Acknowledgments

The authors would like to thank Willem Mallon for his extensive clarifications on Click primitives and asynchronous designs. The XDI state graphs are provided to us by him. This research is supported by NWO/EW projects Formal Validation of Deadlock Avoidance Mechanisms (FVDAM) under grant no. 612.064.811, and Effective Layered Verification of Networks-on-Chip (ELVeN) under grant no. 612.001.108. This research is supported by a grant from Intel Corporation.

References

[1] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click elements: An implementation style for data-driven compilation," in *Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems (ASYNC'10)*, 2010, pp. 3–14.

[2] J. A. Brzozowski and C.-J. H. Seger, *Asynchronous circuits*, ser. Monographs in computer science. Springer, 1995.

[3] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proceedings of the sixth MIT conference on Advanced research in VLSI (AUSCRYPT'90)*. MIT Press, 1990, pp. 263–278.

[4] T. Verhoeff, "Analyzing specifications for delay-insensitive circuits," in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1998, pp. 172–183.

[5] W. Mallon and J. Udding, "Building finite automata from DI specifications," in *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems.*, 1998, pp. 184–193.

[6] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, November 1977, pp. 46–57.

[7] C. Baier, J. Katoen *et al.*, *Principles of model checking*. MIT press, 2008.

[8] M. Kaufmann, P. Manolios, and J. S. Moore, "ACL2 Computer-Aided Reasoning: An Approach," 2000.

[9] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," *Formal Methods in System Design*, vol. 40, no. 2, pp. 147–169, 2012.

[10] W. A. Hunt Jr. and S. Swords, "Centaur technology media unit verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., 2009, vol. 5643, pp. 353–367.

[11] A. Sulflow, U. Kuhne, G. Fey, D. Grosse, and R. Drechsler, "WolFram - a word level framework for formal verification," in *IEEE/IFIP International Symposium on Rapid System Prototyping (RSP'09)*, june 2009, pp. 11–17.

[12] J. Sawada, P. Sandon, V. Paruthi, J. Baumgartner, M. Case, and H. Mony, "Hybrid verification of a hardware modular reduction engine," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*. Austin, TX: FMCAD Inc, 2011, pp. 207–214.

[13] H. Günther, R. Hedayati, H. Loeding, S. Milius, O. Möller, J. Peleska, M. Sulzmann, and A. Zechner, "A framework for formal verification of systems of synchronous components," *Proc. MBEES*, vol. 12, 2012.

[14] C. Yan, F. Ouchet, L. Fesquet, and K. Morin-Allory, "Formal verification of C-element circuits," in *17th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'11)*, april 2011, pp. 55–64.

[15] F. Ouchet, K. Morin-Allory, and L. Fesquet, "Delay insensitivity does not mean slope insensitivity!" in *16th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'10)*, may 2010, pp. 176–184.

[16] M. H. Zaki, S. Tahar, and G. Bois, "Formal verification of analog and mixed signal designs: A survey," *Microelectronics Journal*, vol. 39, no. 12, pp. 1395–1404, 2008.

[17] S. Chatterjee, M. Kishinevsky, and Ü. Y. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification," *IEEE Design & Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[18] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying deadlock-freedom of communication fabrics," in *VMCAI*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 214–231.

[19] F. Verbeek and J. Schmaltz, "Hunting deadlocks efficiently in microarchitectural models of communication fabrics," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'11)*. IEEE, 2011, pp. 223–231.