Verification of Building Blocks for Asynchronous Circuits

Freek Verbeek and Julien Schmaltz
Open University of The Netherlands
Heerlen, The Netherlands
School of Computer Science
{freek.verbeek,julien.schmaltz}@ou.nl

Scalable formal verification constitutes an important challenge for the design of 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 present our approach to using ACL2 to verify necessary and sufficient conditions over asynchronous delay-insensitive primitives. These conditions are used to derive SAT/SMT instances from circuits built out of these primitives. These SAT/SMT instances help in establishing absence of deadlocks. Our verification effort consists of building an executable checker in the ACL2 logic tailored for our purpose. We prove that this checker is correct. This approach enables us to prove ACL2 theorems involving `defun-sk` constructs and free variables fully automatically.

1 Introduction

Today's hardware designs commonly are clocked. A rhythmic clock signal ensures that a designer can assume a discrete notion of time. The clocked design paradigm has many advantages, but they come at a high cost. It induces overhead and delay in terms of speed, data flow and energy [6]. In a clock-free or *asynchronous* design each element acts only when necessary and at its own pace. This can save energy, can increase speed and can decrease latency of communications.

Recently, Click has been proposed as a library for the design of asynchronous circuits [4]. It consists of primitives that are *delay-insensitive*, i.e., primitives that behave correctly regardless of any delay induced by interfacing with the environment. Click primitives are low-level hardware design templates for delay-insensitive elements such as storages, forks, joins and distributors [1]. Connected in a pipelined fashion, the purpose of these primitives is to behave as "lego-like" as possible. They restore a high level of abstraction during the design phase, even when a close link to realistic asynchronous hardware is maintained.

Many state-of-the-art formal verification efforts on asynchronous circuits focus on proving properties over elements in isolation [11, 3, 10]. Deadlock freedom, however, is an emergent 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. Our approach is to automatically derive SAT/SMT instances from Click circuits. If the instance is infeasible, the circuit is deadlock-free. If a solution is found, this solution corresponds to a structural deadlock. This approach has been applied before to synchronous circuits, where it shows great promise in terms of scalability [7, 1].

Consider the network in Figure 1 as an example. 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.

[1] To be more precise, the primitives are quasi-delay-insensitive. For sake of presentation, we do not distinct these terms.

R. Gamboa and J. Davis (Eds.): ACL2 Workshop 2013 (ACL2 '13).
EPTCS 114, 2013, pp. 70–84, doi:10.4204/EPTCS.114.6

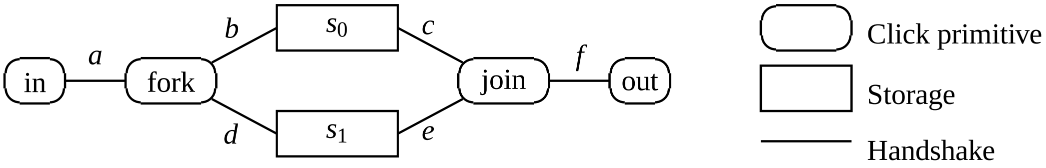

Figure 1: Click circuit

Given this circuit, we automatically derive the following result:

$$\textbf{Dead}(a) \iff ((s_0 \wedge \neg s_1) \vee (s_1 \wedge \neg s_0)) \wedge (s_0 = s_1) \tag{1}$$

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.

Key to deriving a deadlock formula such as Equation 1, is to establish necessary and sufficient conditions for each Click primitive in isolation. These conditions must characterize the reasons that cause a handshake to be *blocked*, i.e., not able to transmit a packet, or *idle*, i.e., not receiving a packet. We use the join as a running example. The join waits for data from its two inputs a and b before forwarding data to output c. Input a is permanently blocked if and only if one of two cases arise. First, when output c is permanently blocked the join can never forward a packet. It therefore blocks input a. Secondly, when no packet arrives at input b, the join will never be able to merge two packets and will never produce an output. Input a is blocked. Hence the join induces the following necessary and sufficient condition:

Complete Condition. *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.*

$$\textbf{Block}(a) \iff \textbf{Block}(c) \vee \textbf{Idle}(b)$$

Correct necessary and sufficient conditions for each primitive in the Click library are vital to the correctness of our approach. Even though their correctness is often seems obvious, their formalizations are complicated and their proofs of correctness are often highly tedious. Moreover, the Click library contains many primitives and our approach requires multiple necessary and sufficient conditions per primitive. Therefore, we have implemented a small and highly tailored checker for Click primitives in ACL2. This checker is able to automatically verify necessary and sufficient conditions built out of block- and idle predicates for a library of delay insensitive primitives. This paper presents ACL2 details of our verification effort, a broader overview can be found in our publication at ASYNC [8]. Details on how these conditions can be used to build a formula such as Equation 1 can be found elsewhere [7, 1].

2 Formalizing Blocking and Idle Conditions

We represent Click primitives using the eXtended Delay Insensitive (XDI) specification [9, 2]. In this paper, XDI specifications are represented using automata. We first introduce the parts of the XDI formalism relevant to this paper. Then, the execution semantics of XDI state machines are formalized. Finally, we use Linear Temporal Logic (LTL) to formalize properties over executions of Click circuits. LTL uses the `G`(lobally) operator to express that some property is always true, and the `F`(inally) operator which expresses that some property is eventually true [5].

2.1 Formalization of Click Primitives

A Click primitive is connected to several other primitives (its *environment*) and may use several handshakes for this. Each handshake is implemented by two wires $h_\mathtt{R}$ and $h_\mathtt{A}$ for requests and acknowledgments. Each wire is either an input to the primitive, or an output. 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.

Running Example, Part 1. *For the join, the set of handshakes is* $\{a,b,c\}$*. The set of input wires is* $\{a_\mathtt{R}, b_\mathtt{R}, c_\mathtt{A}\}$ *and the set of output wires is* $\{a_\mathtt{A}, b_\mathtt{A}, c_\mathtt{R}\}$*. The possibility of transmitting data with requests is not needed.*

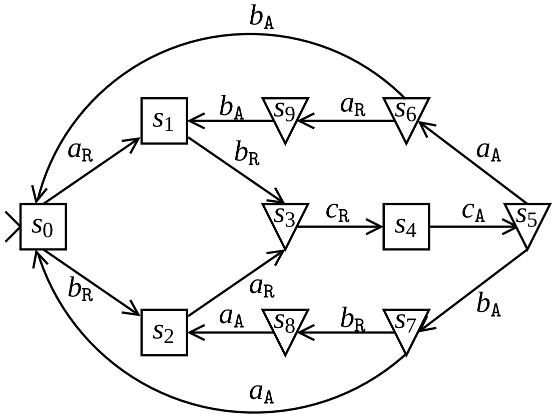

Figure 2: XDI state graph of a join

Figure 2 shows the XDI state machine of the join. The following ACL2 code, which will be explained in more detail hereafter, corresponds to this XDI state graph:

```
(defconst *xdi-sm-join*
 '(;;State Init Type      Transitions
     (s0   T    BOX       (((b R I) s2) ((a R I) s1))))
     (s1   NIL  BOX       (((b R I) s3)))
     (s2   NIL  BOX       (((a R I) s3)))
     (s3   NIL  TRANSIENT (((c R O) s4)))
     (s4   NIL  BOX       (((c A I) s5)))
     (s5   NIL  TRANSIENT (((b A O) s7) ((a A O) s6)))
     (s6   NIL  TRANSIENT (((b A O) s0) ((a R I) s9)))
     (s7   NIL  TRANSIENT (((a A O) s0) ((b R I) s8)))
     (s8   NIL  TRANSIENT (((a A O) s2)))
     (s9   NIL  TRANSIENT (((b A O) s1))))
```

An XDI specification consists of a set of states. There is exactly one state that is the initial state. 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. 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.

Running Example, Part 2. *For the join, state s_0 is the initial state. As this state requires an input from the environment on wires $a_{\mathtt{R}}$ and $b_{\mathtt{R}}$, there are no progress obligations and the type of state s_0 is indifferent. 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, the type of state s_3 is transient.*

A transition `(w s)` is a tuple containing the label `w` that represents the wire on which a communication is to occur for the transition to the next state `s` to happen. A wire `w` is represented by a tuple `(h R/A I/O)` with three values representing the handshake, whether the wire is used for Requests or Acknowledgments, and whether the wire is an input or and output to the primitive. For example, if the join is in its initial state and the input wire $a_{\mathtt{R}}$ changes from low to high, the join moves to state s_1. For details on rules on which transitions are allowed and required in XDI specifications, we refer to papers on the XDI formalism (e.g., [9]).

2.2 Execution Semantics

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. A wire w is stable if and only if its value is permanently unchanged. This implies that if wire w is stable, no transition labelled with w occurs. Therefore, the environment, i.e., the complete set of Click primitives constituting the circuit, is represented as a set of input wires such that each wire in the set is deemed to be stable.

Running Example, Part 3. *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., the environment is the empty set), to an environment where all three input wires of the join are stable (i.e., the environment is the set $\{a_{\mathtt{R}}, b_{\mathtt{R}}, c_{\mathtt{A}}\}$).*

First, we define a predicate to indicate that a given wire is an input wire. We make use of the `proj` function, which returns the *n*th projection in a list of lists. For example, `(proj 0 *xdi-sm-join*)` return the set of states of the XDI state machine of the join.

```
(defun input-wirep (xdi-sm wire)
  (member-equal (list (car wire) (cadr wire) 'I)
                (proj 0 (union (proj 3 xdi-sm)))))
```

Function `input-wirep` takes as input an XDI state machine and a partial description of a wire, namely a tuple `(h R/A)`, where `h` is the handshake and `R/A` indicates a Request or Acknowledge. This is transformed to a wire `(h R/A I)`, for which is searched in the set of labels on the transitions of the XDI state machine. An environment can now be defined as a set of input wires.

```
(defun envp (xdi-sm env)
  (if (endp env)
    t
    (and (input-wirep xdi-sm (car env))
         (envp xdi-sm (cdr env)))))
```

The actual value of the environment depends on the state of the network. In the remainder of this paper, we quantify over *all* possible environments.

Given an environment, we can define the next step function of XDI state machines. Given a current state, this function returns the set of next possible states. First, we define a function that takes as input a list of transitions `ts` and filters out transitions labelled with a stable wire, i.e., transitions that cannot occur. Function `stable` returns `t` if and only if the given wire is stable in the given environment.

```
(defun remove-stable-wire-transitions (ts env)
  (cond ((endp ts)
         nil)
        ((stable (caar ts) env)
         (remove-stable-wire-transitions (cdr ts) env))
        (t
         (cons (car ts)
               (remove-stable-wire-transitions (cdr ts) env)))))
```

This function yields all transitions `((h R/A I/O) s')` that are not stable. The next step function gives it all possible transitions from the current state `s` and takes from each resulting transition the next state `s'`.

```
(defun xdi-step (xdi-sm s env)
  (proj 1 (remove-stable-wire-transitions (nth 3 (assoc s xdi-sm)) env)))
```

2.3 Labelling States as Blocking or Idling

We identify each non-transient state as *blocking* or *idling* with respect to 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 4. *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.*

To define predicates `blocking` and `idling`, we define an executable function `compute-b/i` which recursively explores the state machine and returns an association list mapping to each state a Boolean value indicating how the state should be labelled. The intuition of this function is that initially states are idling. As soon as a transition `((h R I/O) s')` occurs, apparently the primitive is in a state where it has been requested to communicate on handshake `h`, but has not finished this communication yet. All subsequent states are therefore blocking handshake *h*, until a transition `((h A I/O) s')` occurs. After this transition, the primitive has successfully dealt with the request and no communication occurs on handshake *h*. All subsequent states are idling handshake *h*. This repeats, until all states have been explored.

```
(mutual-recursion
 (defun compute-b/i (xdi-sm s h flg ret)
   (if (assoc s ret)
     ret
     (let ((ret (acons s flg ret)))
       (compute-b/i-ts xdi-sm (nth 3 (assoc s xdi-sm)) h flg ret))))
 (defun compute-b/i-ts (xdi-sm ts h flg ret)
   (let ((ret (cond ((equal (caaar ts) h)
                     (compute-b/i xdi-sm (cadar ts) h (not flg) ret))
                    (t
                     (compute-b/i xdi-sm (cadar ts) h flg ret)))))
     (compute-b/i-ts xdi-sm (cdr ts) h flg ret))))
```

Figure 3: Implementation of `compute-b/i`

Figure 3 shows the ACL2 code of function `compute-b/i` which computes this association list. Function `compute-b/i` takes as second parameter a state `s`. The third parameter is a flag indicating whether currently explored states are to be marked blocking or idling. It checks whether this state has already been explored. If so, then no further exploration is needed. Otherwise, it updates the returned association list `ret` by associating the current value of the flag to the current state. After this update, the function recursively explores all transitions leading out of the current state. Function `compute-b/i-ts` takes as second parameter a set of transitions. Sequentially two things occur. First, the first transition of the set is analyzed. If this transition concerns handshake `h`, the flag is changed indicating that a switch from blocking to idling happens, or the other way around. A recursive call with the next state as value for `s` is performed. Second, the remaining transitions are recursively explored.

Function `compute-b/i` is initially called with the initial state and as flag the value `nil`. The predicate `blocking` can now be defined by simply looking up the given state in the result of function `compute-b/i`.

```
(defun blocking (xdi-sm s h)
  (cadr (assoc s (compute-b/i xdi-sm (xdi-get-init-state xdi-sm) h nil nil))))
```

Predicate `idling` is defined as not blocking.

Running Example, Part 5. *The state machine of the join contains a transition* $s_0 \xrightarrow{a_{\mathtt{R}}} s_1$*. We can compute that:*

`(blocking *xdi-sm-join* s0 a)`

evaluates to `nil`*, whereas*

`(blocking *xdi-sm-join* s1 a)`

evaluates to `t`*.*

This represents that state s_0 *is idling handshake a, whereas state* s_1 *is blocking handshake a.*

Remarks

An important assumption on the Click primitives is that their XDI specification ensures that function `blocking` is uniquely defined 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 $h_{\mathtt{A}}$, function `blocking` enforces $\text{blocking}(h,s)$ to be true. If this state s can also be reached with a sequence of transitions without $h_{\mathtt{R}}$ as label, function `blocking` enforces blocking(h,s) to be false. Such state graphs are not allowed. We will call a Click primitive for which function blocking is unique over all non-transient states *unambiguous*. In ACL2, we have an executable function checking for unambiguity.

Function `compute-b/i` does not necessarily terminate. To prove termination, we require both a list of assumptions and some checks which have to be performed by the function before each recursive call. We have added the assumption as guards, and defined a logical version of this function with the additional checks. For the logical version, we have proven termination. The code shown here is the executable version, without these checks. Using an `mbe`-construct, we have proven that under assumption of the guards, the logical and the executable versions are equivalent.

2.4 Formulating Block- or Idle Conditions

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 2). If the environment dictates that wire $c_{\mathtt{A}}$ is stable, any execution will strand 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 a primitive will eventually get stuck in non-transient states labelled “blocking h”.

To define LTL properties over XDI state machines, we first define the notion of trace. A trace is a set of states that is connected via the `xdi-step` function.

```
(defun xdi-tracep (xdi-sm trace env)
  (cond ((endp trace)
         t)
        ((endp (cdr trace))
         t)
        (t
         (and (member (cadr trace) (xdi-step xdi-sm (car trace) env))
              (xdi-tracep xdi-sm (cdr trace) env)))))
```

To express that a machine is permanently stuck in blocking states, we use a `defun-sk` construct to quantify over all possible traces starting in the current state.

```
(defun-sk G-blocking_ (xdi-sm h s env)
  (forall (trace)
          (implies (and (xdi-tracep xdi-sm trace env)
                        (equal (car trace) s))
                   (or (equal (nth 2 (assoc (car (last trace)) xdi-sm))
                              'transient)
                       (blocking xdi-sm (car (last trace)) h)))))
```

The trailing underscore is used to indicate that the function is non-executable. Any trace starting in `s` ends either in a transient state or in a state that is blocking handshake `h`. Note that we deal with finite traces only. Since the XDI automata are always finite, any infinite trace consists of a prefix followed by a repetition of some trace induced by a cycle. It it therefore sufficient to analyze all finite – but of unbounded length – prefixes.

Similarly, we express the `F` operator using a `defun-sk` construct introducing an existential quantifier.

```
(defun-sk F-G-blocking_ (xdi-sm h s env)
  (exists (trace)
          (and (xdi-tracep xdi-sm trace env)
               (equal (car trace) s)
               (G-blocking_ xdi-sm h (car (last trace)) env))))
```

Similar definitions have been formulated for idling. Given environment `env`, handshake *h* is *permanently blocked* if and only if the corresponding XDI state machine is eventually always in a blocking state. Similarly, handshake *h* is *permanently idle* if and only if the corresponding XDI state machine is eventually always in an idling state.

```
(defun Blocked_ (xdi-sm h env)
  (F-G-blocking_ X h (xdi-get-init-state xdi-sm) env))
(defun Idle_ (xdi-sm h env)
  (F-G-idling_ xdi-sm h (xdi-get-init-state xdi-sm) env))
```

Finally, we can formulate necessary and sufficient conditions per Click primitive. For example, the ACL2 formalization of the running example becomes:

Theorem 1.

```
(defthm blocking-equation-join
  (implies (envp xdi-sm env)
           (iff (Blocked_ *xdi-sm-join* 'a env)
                (or (Blocked_ *xdi-sm-join* 'c env)
                    (Idle_ *xdi-sm-join* 'b env)))))
```

3 Model Checking Blocking and Idle Conditions

Proving Theorem 1 could be done manually as follows. First, a case distinction is required over all possible environments (in this case, eight in total). For each environment, all traces have to be explored to see whether the labels computed by function `compute-b/i` always satisfy the formula that is to be proven. As we need multiple theorems per primitive and there is a whole library of Click primitives, we want to prove dozens of theorems such as Theorem 1. Naturally, a manual proof is simply infeasible and an automated approach is mandatory. Therefore, our proof technique is to A.) define executable counterparts to non-executable functions `Blocked_` and `Idle_` and B.) make an automatic enumeration of all possible environments. A once and for all proof that these functions are correctly implemented can then be used to prove Theorem 1 without further interaction.

3.1 ACL2 Overview of Automated Proof

Our objective for Part A. is to implement function `Blocked` in such a way that the following lemma can be proven (similar for `Idle`). This is the specification of function `blocked`, its definition will follow.

Lemma 1.

```
(defthm rewrite-non-exec-Blocked_-to-exec-Blocked
  (implies (and (xdi-smp xdi-sm)
                (member s (proj 0 xdi-sm)))
           (equal (Blocked_ xdi-sm h s env)
                  (Blocked  xdi-sm h s env))))
  :rule-classes :definition)
```

Function xdi-smp recognizes syntactically valid XDI state machines. For any state s that is a valid state, the result of function Blocked is equivalent to that of its specification Blocked_. Once this theory has been established, the non-executable definition Blocked_ is disabled in the theory.

```
(in-theory (disable Blocked_))
```

This way we are sure – when proving a theorem – that any occurrence of Blocked_ will be rewritten using Lemma 1 *only* (note that a defun-sk construct is non-executable, but is still often rewritten to a goal without the original function name, thereby preventing application of Lemma 1).

As for Part B., we straightforwardly implement a function reasonable-envs which takes as input an XDI state machine and generates a list of all possible environments. We first implement function compute-input-wires which given an XDI state machine returns the set of input wires. The set of environments is then computed as follows:

```
(defun reasonable-envs (xdi-sm)
  (powerset (remove-duplicates (compute-input-wires xdi-sm))))
```

The set of input wires is assembled. Duplicate entries are removed, for sake of efficiency. The list of relevant environments contains any subset of these wires.

Remark

The number of environments grows exponentially. The XDI automata of interest are however not very large. In Section 4 we apply our method to a non-trivial Click primitive, namely the distributor. Regardless of the large number of environments, we can easily deal with this primitive.

Using Parts A. and B., we can prove Theorem 1 completely automatically, after rewriting it slightly. Our final formulation becomes:

```
(defthm blocking-equation-join
  (and (xdi-smp-guard *xdi-sm-join*)
       (implies (member env (reasonable-envs *xdi-sm-join*))
                (iff (Blocked_ *xdi-sm-join* nil ’in0 env)
                     (or (Blocked_ *xdi-sm-join* nil ’out env)
                         (Idle_ *xdi-sm-join* nil ’in1 env))))))
```

First, we explicitly verify that the constant *xdi-sm-join* satisfies all guards necessary for correct execution of functions Blocked and Idle. Function xdi-smp-guard is executable and therefore (xdi-smp-guard *xdi-sm-join*) is proven without further interaction. Secondly, we reformulate the theorem, so that open variable env is a member of a computable set of environments. The ACL2 simplifier will compute all reasonable environments. Subsequently, having the following lemma enabled in the theory ensures that the member construct breaks the goal down into eight different subgoals (one for each environment):

```
(defthm member-rewrite
  (equal (member a (cons b x))
         (if (equal a b) (cons b x)
             (member a b))))
```

For each subgoal, the ACL2 simplifier uses Lemma 1 (and a similar lemma for Idle) to rewrite the subgoal to executable versions of Blocked and Idle. At this point, all functions are executable and there are no variables. The truth of each subgoal is automatically evaluated.

3.2 Implementation of `Blocked` and `Idle`

The implementation of `Blocked` needs to check whether eventually generally the machine is in states that are either transient or labelled as "blocking" by function `compute-b/i`. So first a function must be implemented which decides whether in a certain start state `s` the machine generally is in such states, i.e., a function `G-blocking` must be implemented. Figure 4 shows the implementation of this function.

```
(mutual-recursion
 (defun G-blocking (xdi-sm visited s h env)
   (if (not (member-equal s visited))
     (if (or (equal (nth 2 (assoc s xdi-sm)) 'transient)
             (blocking xdi-sm s h))
       (G-blocking-ss xdi-sm (cons s visited)
                      (remove-equal s (xdi-step xdi-sm s env)) h env)
       nil)
     visited))
 (defun G-blocking-ss (xdi-sm visited ss h env)
   (if (endp ss)
     visited
     (let ((ret (G-blocking xdi-sm visited (car ss) h env)))
       (cond ((equal ret nil)
              ret)
             (t
              (let ((ret2 (G-blocking-ss xdi-sm visited (cdr ss) h env)))
                (if ret2
                  (append ret ret2)
                  nil))))))))
```

Figure 4: Implementation of `G-blocking`

Function `G-blocking` takes as input the XDI state machine, an accumulator of visited states, the current state, the handshake and the current environment. If it returns `nil` this indicates that there is some reachable state that is not labelled "blocking" with respect to handshake `h`. If it does not return `nil` it does not return `t`, but instead it returns a list of states that has been explored. This will be used in the proofs later on.

If the current state `s` is not visited and labelled "blocking" with respect to handshake `h`, exploration continues with all next states using function `G-blocking-ss`. This function deals with two cases: either there are no states to be explored, or there are states to be explored. In the first case, as there are no more reachable states, all reachable states have been explored. Therefore, the function should return `t`, but instead returns the accumulator `visited` storing all explored states. Otherwise, the function checks whether the first state in the list is `G-blocking`. If the result of this check is `nil`, this result is returned. Otherwise, the intermediate result (i.e., the states explored in the recursive call) are appended to the final result.

A similar function is implemented to compute `F-G-blocking_.` We extend both functions with an extra flag so that these functions can also be used to compute `(F-)G-idling`. Using these functions, we can define the implementations of `Blocked_` and `Idle_`.

```
(defun Blocked (xdi-sm h env)
  (consp (F-G-blocking xdi-sm nil `(,(xdi-get-initial-state xdi-sm)) h env)))
(defun Idle (xdi-sm h env)
  (consp (F-G-idling xdi-sm nil `(,(xdi-get-initial-state xdi-sm)) h env)))
```

3.3 Proof of Lemma 1

We present the proof of correctness of function `G-blocking`. The proofs for `F-G-blocking` are similar. The proof is in two directions. We prove Lemma 2 which states that for any state such that specification `G-blocking_` returns `t`, executable function `G-blocking` returns a non-empty list of visited states. Secondly, we prove Lemma 3 which states that any state for which executable function `G-blocking` returns a non-empty list, specification `G-blocking_` returns `t`.

Lemma 2.

We formulate the lemma in such a way that induction over `G-blocking` is possible. This requires parameters `visited` and `ss` to be free. Also, any assumption on these variables must be an invariant over function `G-blocking`. Our formalization is as follows, and will be detailed hereafter:

```
(defthm G-blocking_-->G-blocking
  (implies (and (xdi-smp xdi-sm)
                (G-blocking_ xdi-sm h s env)
                (consp ss)
                (subsetp ss (proj 0 xdi-sm))
                (A-xdi-reachable xdi-sm env s ss))
           (G-blocking xdi-sm visited ss h t env))
```

Assume a valid XDI state machine and a state `s` which is generally blocking according to the specification. For any non-empty set `ss` of valid states we prove that executable function `G-blocking` returns a non-empty list. As an invariant, we require that all states in `ss` are reachable from state `s`. Function `A-xdi-reachable` returns `t` if and only if all states in `ss` are reachable from state `s`. Reachability between two states `s1` and `s2` is straightforwardly defined using a `defun-sk` construct as the existence of a non-empty trace starting in `s1` and ending in `s2`:

```
(defun-sk xdi-reachable (xdi-sm env s1 s2)
  (exists (trace)
          (and (xdi-tracep xdi-sm trace env)
               (consp trace)
               (equal (car trace) s1)
               (equal (car (last trace)) s2))))
```

Once it is proven that reachability of states `ss` from state `s` is indeed an invariant, the proof becomes conceptually very easy. As soon as G-blocking encounters a non-transient state `s2`, we know from the invariant that state `s2` is reachable from state `s`. From assumption `(G-blocking_ xdi-sm h s env)` we can prove that `s2` is blocking, which is expressed by lemma `spec-of-G-blocking_`:

```
(defthm spec-of-G-blocking_
  (implies (and (G-blocking_ xdi-sm h s env)
                (xdi-reachable xdi-sm env s s2)
                (not (equal (nth 2 (assoc s2 xdi-sm)) 'transient)))
           (blocking xdi-sm s2 h))
  :hints (("Goal" :use ((:instance G-blocking_-necc
                          (trace (xdi-reachable-witness xdi-sm env s s2)))))))
```

The theorem is proven by instantiating the theorem introduced by the `defun-sk` event corresponding to `G-blocking_`. This instantiation requires a trace from `s` to `s2`. This trace is exactly the witness created by the `defun-sk` event corresponding to `xdi-reachable`.

Now we do induction over `G-blocking`. We have to prove that it does not return `nil`. It returns `nil` only if it encounters an illegal state or a non-transient state that is not labelled “blocking”. The first case cannot happen due to assumption `(xdi-smp xdi-sm)`. The second case cannot happen since the invariant ensures that any explored state is reachable from state `s` and since lemma `spec-of-G-blocking_` can be used to prove that any state reachable from `s` is labelled “blocking”.

Lemma 3.

The lemma is formulated as follows:

```
(defthm G-blocking-->G-blocking_
  (implies (and (xdi-smp xdi-sm (cars xdi-sm))
                (member-equal s (cars xdi-sm))
                (G-blocking xdi-sm nil (list s) h env))
           (G-blocking_ xdi-sm h s env)))
```

Given a valid XDI state machine and a valid state `s`, the executable function correctly decides the LTL formula for state `s`.

For the proof of the lemma we use the fact that function `G-blocking` does not return `t` when all reachable non-transient states are labelled “blocking”, but the accumulator `visited` instead. As we reason over reachable states, we define function `xdi-reach` which takes as input a set of states `ss` and assembles all states reachable from any state in `ss`. The reasoning is as follows:

1. Any state returned by `G_blocking` is either transient or labelled “blocking”.
2. For any set of states `ss`, the set of states accumulated by `(G-blocking xdi-sm visited ss h env)` contains all states returned by `(xdi-reach ss)`.
3. Any state reachable from any state in `ss` is in `(xdi-reach ss)`.
4. We have to prove that any state `s2` reachable from `s` is either transient or labelled “blocking”.
   By 3.) it follows that:
   `(member s2 (xdi-reach (list s))`
   By 2.) it follows that:
   `(member s2 (G-blocking xdi-sm nil (list s) h env)`
   By 1.) it follows that `s2` is either transient or labelled “blocking”.

First, we prove a theorem stating that any non-transient state returned by `G-blocking` is indeed labelled “blocking”.

```
(defthm all-states-in-G-blocking-are-blocking
  (implies (A-blocking xdi-sm visited h)
           (A-blocking xdi-sm (G-blocking xdi-sm visited ss h env) h)))
```

Function `A-blocking` is a universal quantifier expressing that all given states are either transient or labelled “blocking”. Assuming this property holds for all initially accumulated states, this property holds for accumulated states.

We then prove that `G-blocking` returns all states assembled by `xdi-reach`.

```
(defthm all-reachable-states-in-G-blocking
  (implies (G-blocking xdi-sm visited ss h env)
           (subsetp (xdi-reach xdi-sm visited ss env)
                    (G-blocking xdi-sm visited ss h env))))
```

Assuming that `G-blocking` does not return `nil`, we prove that any state that is in the reach of some state in `ss` is also member of the list of accumulated states returned by `G-blocking`.

Finally, we prove correctness of function `xdi-reach`, i.e., that it contain all reachable states.

```
(defthm spec-of-xdi-reach
  (implies (and (xdi-smp xdi-sm (proj 0 xdi-sm))
                (member s (proj 0 xdi-sm))
                (xdi-reachable xdi-sm env s s2))
           (member-equal s2 (xdi-reach xdi-sm nil (list s) env)))
```

Using these lemmas, Lemma 3 can be proven without induction. To prove the universal quantifier introduced by `G-blocking_`, we have to prove of a witness state

$$s_w = \texttt{(G-blocking\_-witness xdi-sm h s env)}$$

that it is either transient or labelled “blocking”. Instantiating the first lemma with the accumulator `visited` set to `nil`, automatically discharges its assumption. What remains to be proven is that s_w is accumulated by `G-blocking`. This is proven by the second lemma, instantiated with `visited` set to `nil` and `ss` set to `(list s)`. This forces us to prove that state s_w is a member of `xdi-reach`. The third lemma is used to prove this, instantiating `s2` with `(G-blocking_-witness xdi-sm h s env)`.

4 Application

The *distributor* (see Figure 5) is a Click primitive used for routing packets through a network. It uses handshake a on which the availability of data is communicated and three 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. Figure 6 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.

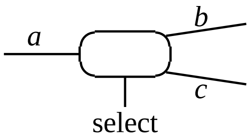

Figure 5: Schematic overview of the distributor

The blocking equation of input `a` of the distributor is shown in Figure 7. Blockage of input `a` is logically equivalent to three cases. First, if no select signal ever arrives at input `select`, the distributor will not know how to route packets and will therefore not transmit them. Secondly, if always eventually

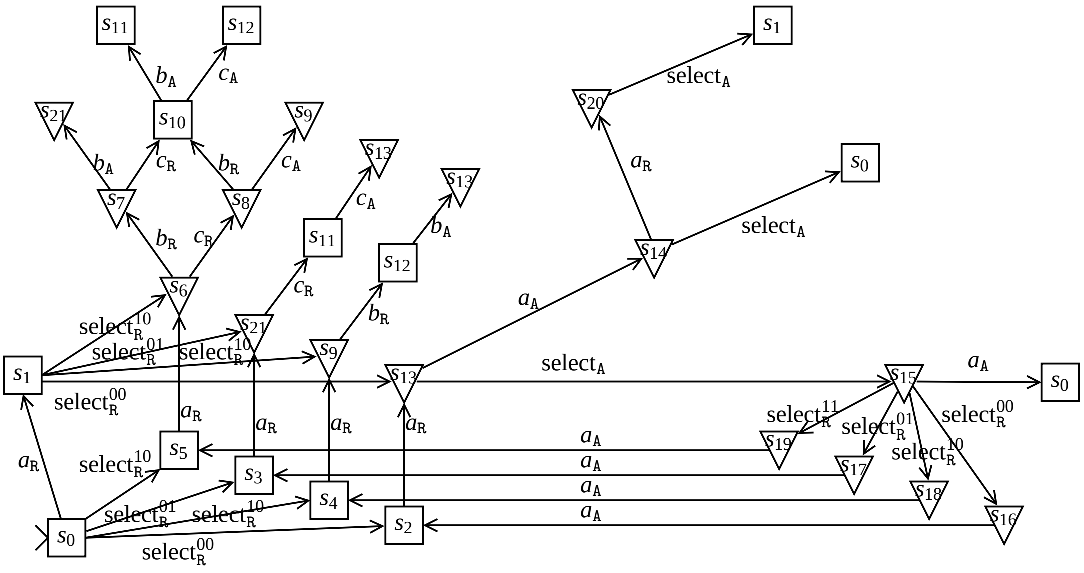

Figure 6: XDI state graph of the distributor. Note that several states (e.g. s_{11} and s_{12} are replicated in order to simplify the diagram.

a 01 signal arrives on the `select` wire, and if output `b` is permanently blocked, eventually a packet at input `a` will be permanently blocked (note that a 01 signal on the `select` wire means "route towards output `b`"). The third case is similar but for output `c`.

Using the lemmas presented in the previous section, theorem in Figure 7 can be proven instantaneously and without any interaction.

5 Conclusion

We have mechanically verified properties of a library of delay-insensitive primitives in the ACL2 theorem prover. These properties are often deceptively simple, making it easy to formulate incorrect theorems. Moreover, their proofs are large and cumbersome. Their formalization is tricky: it is based on XDI state

```
(defthm blocking-equation-distributor
  (and (xdi-smp-guard *xdi-sm-distributor*)
       (implies (member-equal env (reasonable-envs *xdi-sm-distributor*))
                (iff (Blocked_ *xdi-sm-distributor* 'in env)
                     (or (and (Idle_ *xdi-sm-distributor* 'select00 env)
                              (Idle_ *xdi-sm-distributor* 'select01 env)
                              (Idle_ *xdi-sm-distributor* 'select10 env))
                         (and (not (Idle_ *xdi-sm-distributor* 'select01 env))
                              (Blocked_ *xdi-sm-distributor* 'out1 env))
                         (and (not (Idle_ *xdi-sm-distributor* 'select10 env))
                              (Blocked_ *xdi-sm-distributor* 'out0 env)))))))
```

Figure 7: Blocking Equation for Distributor

machines and their execution semantics relative to the environment of the primitive. Our approach consists of building a checker for XDI state machines which can decide LTL formulae that are built out of block- and idle predicates. This checker has been proven correct with respect to its specification. The theorems that are to be proven involve free variables and non-executable functions introduced by `defun-sk` constructs. Loading the book that contains our definitions and lemmas suffices to fully automatically prove these theorems quickly.

The properties that have been proven are used to derive a SAT/SMT instance from an asynchronous circuit built out of primitives in the library. This derivation can be quite contrived, especially when data is taken into account. In the future, we plan to use ACL2 to prove correctness of our derivation, proving that feasibility of the derived SAT/SMT instance is logically equivalent to the existence of a structural deadlock.

References

[1] Alexander Gotmanov, Satrajit Chatterjee & Michael Kishinevsky (2011): *Verifying Deadlock-Freedom of Communication Fabrics*. In Ranjit Jhala & David A. Schmidt, editors: *VMCAI*, *Lecture Notes in Computer Science* 6538, Springer, pp. 214–231. Available at `http://dx.doi.org/10.1007/978-3-642-18275-4_16`.

[2] W.C. Mallon & J.T. Udding (1998): *Building finite automata from DI specifications*. In: *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems.*, pp. 184–193, doi:`10.1109/ASYNC.1998.666504`.

[3] F. Ouchet, K. Morin-Allory & L. Fesquet (2010): *Delay Insensitivity Does Not Mean Slope Insensitivity!* In: *16th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'10)*, pp. 176–184, doi:`10.1109/ASYNC.2010.27`.

[4] A. Peeters, F. te Beest, M. de Wit & W. Mallon (2010): *Click elements: An implementation style for data-driven compilation*. In: *Proceedings of the IEEE Symposium on Asynchronous Circuits and Systems (ASYNC'10)*, pp. 3–14, doi:`10.1109/ASYNC.2010.11`.

[5] Amir Pnueli (1977): *The temporal logic of programs*. In: *18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, doi:`10.1109/SFCS.1977.32`.

[6] Ivan Sutherland (2012): *The tyranny of the clock*. *Communications of the ACM* 55(10), pp. 35–36, doi:`10.1145/2347736.2347749`.

[7] F. Verbeek & J. Schmaltz (2011): *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, pp. 223–231.

[8] Freek Verbeek & Julien Schmaltz (2013): *Formal Deadlock Verification for Click Circuits*. In: *19th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'13).*

[9] T. Verhoeff (1998): *Analyzing specifications for delay-insensitive circuits*. In: *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 172–183, doi:`10.1109/ASYNC.1998.666503`.

[10] Chao Yan, F. Ouchet, L. Fesquet & K. Morin-Allory (2011): *Formal Verification of C-element Circuits*. In: *17th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'11)*, pp. 55–64, doi:`10.1109/ASYNC.2011.14`.

[11] Mohamed H. Zaki, Sofine Tahar & Guy Bois (2008): *Formal verification of analog and mixed signal designs: A survey*. *Microelectronics Journal* 39(12), pp. 1395–1404, doi:`10.1109/IDT.2009.5404084`.