Catapult HLS User View Scheduling Rules

Stuart Swan, Platform Architect, Siemens EDA

20 February 2026

Introduction

In a high-level synthesis (HLS) design methodology, much of the benefit is lost if design verification and debug must still be performed at the RTL level. In manual RTL design flows, RTL verification is typically one of the most costly and time-consuming phases of the overall flow. In HLS flows, RTL-level verification is even more challenging, because the RTL is machine-generated rather than hand-crafted.

This document presents a set of HLS scheduling rules, a modeling methodology, and a collection of formal proofs that together allow almost all design verification and debug for full systems to be carried out on the pre-HLS SystemC model. The methodology and proofs are organized around a latency-insensitive design style, while still fully supporting real-world techniques such as:

- latency-sensitive signal-level protocols
- latency-sensitive portions of systems
- both sequential and combinational logic
- pipelined designs
- shared memories between sequential processes
- reordering of RAM accesses to optimize hardware pipelines
- removal of pipeline registers for stable signal inputs to hardware pipelines

These techniques capture the requirements gathered from hundreds of Catapult HLS customer tape-outs and are needed to meet the demanding quality-of-results (QOR) targets traditionally achieved with manual RTL design.

The methodology presented here builds on established verification practices such as SystemVerilog UVM, constrained-random stimulus generation, and functional and code coverage. The system under test is modeled and verified in SystemC, and formal proofs establish behavioral equivalence for the synthesized RTL implementation.

Although the focus of this document is HLS, the design methodology and formal proofs can also be applied when HLS is not used, enabling much more efficient verification and debug at a higher level of abstraction than RTL.

A document abstract is provided in Appendix M.

Document Goals

This document presents HLS tool scheduling rules and modeling methodology rules. The goals are:

1. To provide easy-to-understand rules to HLS users.
2. To ensure precise and consistent rules in both SystemC and C++.
3. To offer rules that are effectively compatible with how Catapult currently operates.
4. To enable the best possible *quality of results* (QOR) in Catapult synthesis.

5. To cover all known user requirements and scenarios.
6. To serve as a suitable starting point for a standardization proposal (e.g., in Accellera SWG).

To illustrate the goals of this document more specifically, consider what an engineer writing a testbench for an HLS model needs to understand about how HLS tools operate. This engineer may be using SystemVerilog UVM or may be writing a testbench in C++/SystemC. They likely are not an expert in any specific HLS tool (and may not want to be), but their testbench needs to work for both the pre-HLS model as well as the post-HLS model. Thus, it is crucial for the DV engineer to have a precise understanding of how the HLS tool will transform the design while still enabling it to be fully verified. This document describes what transformations the HLS tool is allowed to perform so that the pre-HLS and post-HLS models can be effectively verified with the same testbench. The overarching philosophy of the scheduling rules is to present "no surprises" to such a DV engineer, while still giving the HLS tool ample freedom to optimize the design.

Background

HLS tools generate RTL from C++ models. Broadly speaking, this conversion takes a sequential C++ model and turns it into concurrent hardware that maintains the same behavior. HLS tools identify concurrent processes within the C++/SystemC model and then independently synthesize each process. Briefly, some of the techniques that HLS tools use to achieve good HW QOR when synthesizing each process include:

- Optimized scheduling based on the selected silicon target technology.
- Automatic HW pipeline construction according to the user's specifications.
- Automatic HW resource sharing.
- Automatic scheduling of memory accesses.

The internal behavior of each process is specified by the control and dataflow behavior of the C++ code within the process. However, the external communication that each process has with other processes and HW blocks is specified via IO operations that are coded within the model. To enable a reliable, scalable, and verifiable HLS flow that generates high quality hardware, the scheduling behavior of these IO operations needs to be precisely handled at all steps of the flow. This document specifies the rules that govern the scheduling behavior for these IO operations within HLS models. These rules are specified with respect to an individual process, but the intent of the rules is to enable reliable and verifiable behavior of large sets of interacting processes operating as a system in real-world designs. (Appendix G provides formal guarantees regarding the equivalence of the pre-HLS and post-HLS systems.)

By default, HLS tools can insert additional clock cycles (or latency) anywhere within a process -- for example, when pipelining a loop or to enable HW resource sharing. The overall approach used in this document is to make the entire design and testbench system *latency insensitive* to the maximum extent possible, while still fully enabling key HW optimizations. In addition, the overall approach enables the pre-HLS system simulation to be capacity-accurate and throughput-accurate with respect to the post-HLS RTL system.

In some cases, designs or testbenches may use protocols which are latency sensitive. These situations can be handled by isolating the latency sensitive portions to small, self-contained parts of the design or

testbench, and then keeping the rest of the design and testbench latency insensitive. See Appendix D for more information.

In many cases the overall design will need to satisfy end-to-end latency requirements. For these designs it is still highly advantageous to use a latency-insensitive modeling approach and verify in the post-HLS model that the overall design latency requirements have been satisfied, since this is typically easy to do.

This document focuses on sequential HW processes. Combinational HW processes are supported but mostly not discussed since their synthesis is straightforward. All the examples and discussion in this document are for SystemC processes that are sensitive only to a single rising clock edge. (Appendix O discusses support for multiple clock domains.)

This document distinguishes between the following:

1. The *conceptual model* for the scheduling rules.
2. The simulation behavior of a model using the rules in C++ or SystemC.
3. The synthesis of a C++/SystemC model using the rules in an HLS tool such as Catapult.

The goal is to align each of these three cases as closely as possible, so that the user has easy to understand rules, while simulation and synthesis work without surprises. However, as we will see, there are practical considerations which may in certain cases cause small deviations from the conceptual model in either simulation or HLS.

A simple real-world example that illustrates the motivation for the scheduling rules is provided in Appendix A of this document.

The examples referred to in this document are available here:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/tree/master

The most up-to-date version of this document is available here:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/doc/catapult_user_view_scheduling_rules.pdf

An Analogy from RTL Synthesis

To better understand the specific purpose of this document, let's consider how RTL synthesis works. Say you have a sequential block that you are modeling in Verilog RTL, and it has an output port coded like:

```
Out1 <= #some_delay new_val;
```

In Verilog simulation, if some_delay is less than the clock period of the block, then it will probably not affect the overall cycle level behavior of the system during simulation. However, if some_delay is more than the clock period, it probably will.

During RTL synthesis, all RTL synthesis tools will ignore all delays in the input model, in this case even if some_delay is greater than the clock period. Some RTL synthesis tools might give a warning or error for the code above like "Simulation and synthesis results are likely to mismatch because the delay in model is greater than clock period." Some RTL synthesis tools might outright reject a model containing such delays.

One might argue that RTL synthesis tools should always match the Verilog simulation behavior of the input model. But the overall approach works well because RTL is a good and simple *conceptual model* that users and tool vendors can align around. The slight differences between the *simulation model* and the *conceptual model* used by RTL synthesis tools can be easily managed.

We'll return to this example later in this document.

Next, let's clarify some terminology related to signal IO. Say you have a Verilog model like:

```
forever begin
  @(posedge clk);  // wait for 1st rising clock edge
  output1 <= input1 + 10;
  @(posedge clk);  // wait for 2nd rising clock edge
end
```

In this example, input1 is sampled when the process wakes on the first @(posedge clk) and evaluates the RHS. For the purposes of the scheduling rules, we anchor that Read(input1,·) to the most recent synchronization point. The resulting Write(output1,·) is anchored to the next synchronization point (the cycle in which the written value is treated as stable for others to observe).
Cycle-level anchoring convention (used throughout this document). For cycle-level reasoning, each signal IO is assigned an anchor cycle relative to surrounding synchronization calls (e.g., wait() / SyncChannel / @(posedge clk)):
• Read(sig, val) is anchored to the closest preceding synchronization call, i.e. pred_S(Read).
• Write(sig, val) is anchored to the closest succeeding synchronization call, i.e. succ_S(Write).
This convention is only for cycle-level reasoning and does not depend on simulator update regions or delta-cycle ordering.

Catapult HLS Status Concerning Rules in this Document

A separate document provides a list of clarifications related to support within Catapult HLS for the rules described in this document. The items in the list are named *Cat#*, so that each item has a unique number.

This document annotates certain rules with *Cat#* to refer to the items in the separate document.

Terms Used in this Document

Latency-Insensitive: In digital hardware design, *latency-insensitive* refers to a system that operates correctly despite variable communication delays between components. This is achieved by using mechanisms to decouple computation from communication timing. Such designs improve scalability and reliability in complex systems with unpredictable or variable latencies. ARM's AXI4 and APB are examples of latency-insensitive protocols. ARM's AMBA 5 CHI credit-based NOC protocol is also an example of a latency-insensitive protocol.

Process: In Verilog, a process is an *always block* and its equivalent constructs. In SystemC, a process is an instance of an SC_THREAD or SC_METHOD.

Message-passing Interface: A *message-passing interface* reliably delivers messages (or transactions) from one process to another. This document uses this term to denote the type of communication found in Kahn Process Networks. See https://en.wikipedia.org/wiki/Kahn_process_networks
Message-passing read interfaces are always separate from message-passing write interfaces – there are no bidirectional message-passing interfaces. In this document, message-passing channels are assumed to be point-to-point: for each channel c, exactly one producer performs all Push_c operations and exactly one consumer performs all Pop_c operations.

Synchronization Interface: A *synchronization interface* synchronizes one process with another and/or with a global clock. For an example of a synchronization interface, see https://en.wikipedia.org/wiki/Barrier_(computer_science)

Signal IO: In digital HW design, signals are the fundamental communication mechanism. Signals enable communication between two HW blocks/processes, but communication with signals in real HW always incurs at least some delay because communication cannot be faster than the speed of light.
In HDLs and in SystemC, signal delays are modeled with the *delayed update* semantics.

blocking / non-blocking: In this document, a blocking message-passing operation is modeled as issuing a persistent request that remains asserted (and with stable payload, if applicable) until the operation commits (the message is sent or received). A non-blocking message-passing operation returns immediately with a status indicating whether the operation committed in that cycle.

One-way / two-way handshake protocols: A one-way handshake protocol only has one signal between a sender and receiver to synchronize communication. A two-way handshake protocol has two signals (one in each direction) between a sender and receiver to synchronize communication.

shall: This term indicates that a compliant tool or flow is required to follow the indicated rule.

may: This term indicates that a compliant tool or flow is allowed to follow the indicated rule but is not required to do so.

Classes of Operations Involved in Scheduling Rules

There are three classes of operations involved in the scheduling rules:

1. Calls to message-passing interfaces (which are all ac_channel methods, all SystemC MIO calls except calls to SyncChannel)
2. Calls to synchronization interfaces (which are calls to ac_sync and Matchlib SyncChannel, also ac_wait and SystemC wait)
3. Signal IO (which are SystemC signal reads and writes, also C++ model *direct inputs*)

All the operations above are referred to as *IO operations*.

Basic Conceptual Model

The basic conceptual model encompasses processes that have no loop pipelining but may have preserved loops. If a process has a preserved loop, then the user may place a wait statement in the loop. Wait statements explicitly placed in the model are called *explicit wait statements* in this document and are classified as calls to a *synchronization interface*.

The basic conceptual model rules are:

1. Synchronization interface calls within a process always remain in the source code order.

2. Signal read operations occur at the closest preceding call to a synchronization interface. (Cat1)

3. Signal write operations occur at the closest succeeding call to a synchronization interface.

4. Message-passing operations are free to be reordered subject to the following constraints:
   - All message-passing operations before a call to a synchronization interface shall be issued before or in the same cycle in which the synchronization interface commits. (Here "the synchronization interface commits" refers to the commit of that synchronization call in the calling process's trace. For blocking message-passing operations (Push/Pop), the process cannot complete the synchronization call until any earlier issued blocking message requests in that interval have committed.)
   - All message-passing operations after a call to a synchronization interface shall be issued after the cycle in which the synchronization interface call commits.
   - Two message-passing operations on separate interfaces which appear in sequence in the model may be issued either in the same sequence or in parallel in simulation and in synthesis, but they shall not be issued in the reverse sequence. (Cat2)

Some explanation for the very last point: While pure message-passing systems with unbounded FIFOs are immune to reordering concerns, real-world hardware implementations have finite buffer sizes. Reversing the order of message-passing calls in a system with bounded FIFOs can introduce deadlocks that were not present in the original model. The last rule prevents HLS from introducing deadlocks *via this specific mechanism* (reversing the program order of message-passing calls that appear in sequence in the source). However, deadlock behavior can still be affected by other transformations that change the *timing* of request/commit—most notably overlapped execution in pipelined loops and finite-capacity effects.

Pipelined Loops

When a loop is pipelined in HLS, the body of the loop is split into pipeline stages. HLS may start the next iteration of the loop before the current iteration has completed, thus increasing throughput as compared to a non-pipelined implementation.

The user may place wait statements in the body of a pipelined loop to manually separate operations into their respective pipeline stages but is not required to (and in most cases will not do so). We call these wait statements *explicit pipeline stage wait statements*. The scheduling rules for pipelined loops are the same as the rules given in the basic conceptual model, with the addition of these pipeline stage wait statements into the set of calls to synchronization interfaces.

To guarantee equivalent behavior between the pre-HLS and post-HLS systems, pipelined loops must be implemented with automatic flush to prevent deadlocks. See Appendix B for a formal presentation of pipelined loops.

When HLS pipelines a loop, multiple iterations of the loop are overlapped and execute at the same time. During loop pipelining, for all IO operations, HLS shall ensure that an access to a specific message-passing interface, signal, or synchronization interface shall not be reordered relative to same-interface accesses from other iterations.

During pipelining any signal reads and writes and associated synchronization interface calls become embedded in specific pipeline stages. Considering the entire set of signals read or written by the process, if HLS pipelining would cause the order of signal IO to differ between the pre-HLS and post-HLS models, or if any two signal IO operations that occur on the same clock edge in the pre-HLS model would not occur on the same clock edge in the post-HLS model, then the HLS tool shall detect such a situation and require the user to explicitly indicate in the model source (e.g., via a pragma) that HLS pipelining can still be used. (Cat7) In other words, if a process communicates with other processes using only latency-insensitive protocols, then HLS pipelining by default shall not break any of the protocols.

Direct Inputs

Normal SystemC signal read operations occur at the closest preceding synchronization interface call (e.g., wait statement) in both the pre-HLS and post-HLS models. (Cat1). If the HLS tool adds states to the process, or if it pipelines the process, then this implies that the HLS tool must add registers for each such read operation such that the read occurs where specified in the pre-HLS model, and the value is stored until the point where it is consumed in the post-HLS model. The area cost of such registers may be high if there are a lot of signals, and in some cases, it may be unneeded area since the value of the signals may not actually need to be stored internally to the process.

The simplest case is if such external signals are held stable after the process comes out of reset. In this case, HLS may assume that it is free to read the signal values as late as possible, with no need for register storage. This case is handled with the following pragma on SystemC signals and ports (Cat6):

#pragma hls_direct_input

To ensure that there are no pre-HLS versus post-HLS simulation mismatches, the environment that drives the signal shall hold it stable after all receiving processes that use this signal with this pragma come out of reset. Note that with this approach it is allowable to have *dynamic resets*, i.e. activation of block resets and associated resetting of direct input signals as part of the normal operation of the HW.

A related but somewhat more complex case is where input signals to the HW block may only be changed at “agreed upon” times, typically while the portion of the HW block that relies on them is temporarily idle. For example, a block may process 2D images. At the start of each new image, it may be desirable for the TB or external environment to update the control signals for how the block will process the next image. Typically HLS designs are pipelined, and the HW pipeline for the current iteration must be fully *ramped down* before the input signals can be updated to affect the next iteration. In this case we can use the SystemC *SyncChannel* or C++ *ac_sync* primitives to precisely synchronize the DUT with the TB/environment to enable the input signals to be updated at the correct time. The `#pragma`

`hls_direct_input_sync` directive shown below associates the sync operation with the direct inputs that it controls. The precise synchronization scheme shown here ensures that there are no pre-HLS versus post-HLS simulation mismatches even though we are using direct inputs and also changing their values while the design is executing.

```
  // This is example 61* in Catapult Matchlib examples
  sc_in<bool> SC_NAMED(clk);
  sc_in<bool> SC_NAMED(rst_bar);

  Connections::Out<uint32_t>  SC_NAMED(out1);
  Connections::In<uint32_t>   sample_in[num_samples];
  Connections::SyncIn         SC_NAMED(sync_in);
#pragma hls_direct_input
  sc_in<uint32_t> direct_inputs[num_direct_inputs];


  void main() {
    out1.Reset();
    sync_in.Reset();

#pragma hls_unroll yes
    for (int i=0; i < num_samples; i++) {
      sample_in[i].Reset();
    }

    wait();  // reset state

    while (1) {
#pragma hls_direct_input_sync all
     sync_in.sync_in();

#pragma hls_pipeline_init_interval 1
#pragma hls_stall_mode flush
     for (uint32_t x=0; x < direct_inputs[0]; x++) {
       for (uint32_t y=0; y < direct_inputs[1]; y++) {
        uint32_t sum = 0;
#pragma hls_unroll yes
        for (uint32_t s=0; s < num_samples; s++) {
           sum += sample_in[s].Pop() * direct_inputs[2 + s];
        }
        ac_int<32, false> ac_sum = sum;
        ac_int<32, false> sqrt = 0;
        ac_math::ac_sqrt(ac_sum, sqrt);  // internal loop unrolled in cat .tcl file
        if (sqrt > direct_inputs[7])
          out1.Push(sqrt);
       }
     }
    }
  }
};
```

From the perspective of the testbench or the environment, the updating of the signals controlled by the `hls_direct_input_sync` directive shall only occur at a precise point. The TB shall first wait for the `rdy` signal for the sync to be asserted by the DUT, and then the TB shall update all the input signals it wishes to change while simultaneously driving the sync `vld` signal high for one cycle.

It is important to note that the only safe operation to use to synchronize the updating of direct inputs is the sync operation as shown above. Other operations such as Push/Pop or ac_channel operations should not be used for this. In terms of the Appendix G trace model, the Σ-visible Read(sig,val) events

remain anchored to their synchronization points; any additional internal sampling made by the implementation between anchors is treated as an ε-step and must not change the recorded value label.

In the example above the DUT block that is being synthesized determines when to call sync, and thus it determines when the direct inputs will be updated. In some cases, it may be necessary for the environment around the DUT to determine when the direct inputs should be updated. In this case the same approach as shown above should be used, however a separate input from the environment to the DUT (either using a signal or a message-passing interface) should request that the DUT call sync as soon as feasible. This will ensure that the DUT has properly ramped down its pipeline and is ready to receive the newly updated direct inputs as per the overall synchronization scheme described above.

Additional Options for Scheduling Message-passing Interfaces

The following option may be added during HLS (Cat2):

```
STRICT_IO_SCHEDULING=relaxed
```

When this is specified, the HLS tool is allowed to reorder message-passing interface calls freely on distinct interfaces (still not moving calls across synchronization interface calls). This relaxed mode may violate the default no-reverse discipline and therefore may introduce deadlocks that are not present under the default rules. It is recommended that this relaxed option only be used when the user wants to see what order the HLS tool prefers to schedule message-passing interface calls (e.g., to achieve best QOR).

Once the user knows the preferred order, it is recommended that the user modify the pre-HLS source code to reflect the preferred order, and then return to the use of the default scheduling modes. This methodology ensures that HLS cannot introduce new deadlocks *via reversal of source order on distinct message interfaces* as described earlier. Deadlock behavior may still depend on bounded-capacity effects and on overlapped execution (e.g., loop pipelining); those cases are addressed separately in this document via the pipelined-loop discussion (automatic flush / WFG-inertness) and the later deadlock reasoning. The formal equivalence and deadlock-preservation results in Appendices G–K assume STRICT_IO_SCHEDULING is not set to relaxed.

Scheduling of Array Accesses

Arrays may appear in HLS models, and they may be preserved through synthesis and mapped to RAMs. Pointers may also appear in HLS models, and pointer dereferences are resolved to array accesses during HLS.

There are two cases to consider for arrays for the purposes of the scheduling rules:

1. Array instantiation in the HW is internal to the process
   - The array accesses are not visible external to the process, and thus their scheduling is also not visible externally.
   - All the scheduling rules described elsewhere in this document remain unaffected in this case.

2. Array instantiation in the HW is external to the process

- In this case the user model shall indicate how array accesses are mapped onto IO operations that are external to the process.
- We call this the *array access mapping layer*. The *array access mapping layer* maps array accesses onto IO operations described above (signal IO, message-passing interface calls, and synchronization calls).
- The user model may indicate that it is allowable for HLS to transform array accesses, for example, to cache, merge, split, or reorder array accesses (e.g., to improve QOR). These transformed operations, if allowed, are an outcome from the use of the *array access mapping layer*.
- In all cases the scheduling rules described elsewhere in this document for the core IO operations (signal IO, message-passing calls, synchronization calls) remain unaffected.
- In all cases array accesses shall remain constrained by any explicit synchronization interface calls present in the process in the source model. Precisely speaking, all array reads or writes before a synchronization call shall commit before or in the same cycle at which the synchronization call commits, and all array reads or writes after a synchronization call shall commit in a cycle after the synchronization call commits. This rule is stated in terms of commit cycles (the cycle the mapped memory storage is accessed); the scheduler enforces it by choosing issue cycles consistent with the fixed access latency defined by the array access mapping layer.
- Note that if transformed operations occur and array accesses are visible externally in both pre-HLS and post-HLS model, then comparison of pre-HLS and post-HLS behaviors may need to account for the transformed operations.

When a loop is pipelined, multiple iterations of the loop are overlapped and execute at the same time. During loop pipelining, an access to a memory interface (or array) may be moved over or in parallel with an access to the same memory if the HLS tool can prove the reordering is conflict free. If the array/memory is external to the process, the *array access mapping layer* shall indicate that such reordering is allowable if such reordering is to occur during loop pipelining.

Definitions: Issue vs. Commit

Informally, we can define commit and issue as follows:

Definition: Commit (message channel) — as observed by an external clocked observer
Commit is the cycle in which an external clocked observer sees a transfer *complete*:

- A message commits in the cycle when the observer sees both valid and ready high at the same time.
- The committed payload is the data value seen in that same cycle.

Definition: Issue (message channel) — as observed by an external clocked observer
Issue is the cycle in which an external clocked observer sees a transfer *start*:

- A message action issues when a Push operation asserts vld, or a Pop operation asserts rdy.

See Appendix C for formal definitions of issue and commit.

For RAM reads/writes as presented above, commit denotes the precise cycle the memory storage is accessed; since the access latency is fixed and known to the HLS scheduler, it can schedule the operation's issue cycle so that the commit cycle satisfies the required ordering relative to Sync. The "Scheduling of Array Accesses" constraints above are therefore written in commit-time.

Additional States Added by HLS Synthesis

By default, HLS synthesis tools may add additional states to processes (e.g. add latency to enable resource sharing), which may introduce latency differences in the interface behavior between the pre-HLS and post-HLS models. These additional states are never included in the set of *synchronization interface calls* as described above.

When the directive IMPLICIT_FSM=true is set on a process, the HLS synthesis tool shall ensure that the cycle level behavior of the interfaces of the pre-HLS and post-HLS models shall be identical. With this option, the internal state machines of the pre-HLS and post-HLS models will be the same.

When the directive IO_MODE=FIXED is set on a process, the HLS synthesis tool shall ensure that the cycle level behavior of the interfaces of the pre-HLS and post-HLS models shall be identical. With this option, it is still possible that the state machine internal to the process is different between the pre-HLS and post-HLS models (e.g. the post-HLS model may choose to use a pipelined multiplier where the pre-HLS model did not.)

Avoiding Pre-HLS and Post-HLS Simulation Mismatches

The scheduling rules described in this document are designed to be easy to understand, while providing good QOR via HLS and generally avoiding any mismatches between the pre-HLS and post-HLS simulation behaviors.

Non-blocking message-passing operations (PushNB/PopNB in SystemC, C++ ac_channel nb_read/nb_write) are a potential source of mismatches between pre-HLS and post-HLS simulations since their behavior is inherently dependent on the latency within the model, which often changes during HLS. Because of this, non-blocking message-passing interfaces should only be used when no alternative approach is possible. For example, non-blocking message-passing interfaces are required to model time-based arbitration of multiple message streams which access a shared resource. A full discussion of recommended guidelines on the use and verification of non-blocking message-passing interfaces is provided in Appendix L. Note that the scheduling rules described previously in this document fully specify how HLS tools are required to schedule such operations.

Unidirectional message-passing between two processes should not be relied on to achieve synchronization between the two processes, since in general the message latency and storage capacity between the processes may be variable. Also, depending on buffering/pipelining in the realization, the time between a producer's committed Push and the consumer's committed Pop (and the transient in-flight occupancy) may vary. Two unidirectional message-passing channels in opposite directions can be relied upon for synchronization between two processes. (An example of this is the AXI4 ar and r, and aw and b, channels). Note that such synchronization is weaker than explicit synchronization like SyncChannel or signal IO that uses a two-way handshake.

SystemC signal IO operations are a potential source of pre-HLS versus post-HLS simulation mismatches since timing behaviors may change between the two models. The following section provides guidance and rules to help avoid potential mismatches due to signal IO.

When signal IO operations are synchronized with a wait statement, there generally should be a proper two-way handshake associated with the wait statement so that the signal IO is latency insensitive. (Note that this statement does not apply to the signal synchronization approaches described in the Direct Inputs section.)

For example, here's a simple two-way handshake protocol when writing the signal `out_dat`:

```
out_dat = value;
out_vld = 1;
do {
  wait();
} while (out_rdy != 1);
out_vld = 0;
```

And here's a two-way handshake example when reading signal `in_dat`:

```
in_rdy = 1;
do {
  wait();
} while (in_vld != 1);
value = in_dat;
in_rdy = 0;
```

Some signal-level protocols have different two-way handshaking approaches (e.g. ARM APB), but they are still latency insensitive.

If signal IO operations are associated with a wait statement and that wait statement does not have a proper two-way handshake, then the signal IO is likely to be latency sensitive and may result in pre-HLS versus post-HLS simulation mismatches. In some systems a one-way signal handshake is sufficient for reliable system operation. See Appendix E for further discussion.

The scheduling rules state that signal IO operations occur at either SystemC wait statements or SyncChannel calls (sync_in and sync_out). In the remainder of this section, we will use *wait* statements to refer to both.

RULE 1: It is always best coding style to group signal write operations just before their corresponding wait statement, and signal read operations just after their corresponding wait statement. (Cat3). An example is below:

```
 sc_in<int> i1;
 sc_in<bool> go;
 sc_out<int> o1;
 void my_thread() {
   int new_val=0;
   while (1) {
     o1.write(new_val);
     do {
       wait();
     } while (!go.read());
     new_val = i1.read();
     new_val = some_function(new_val); // function has no internal IO
   }
}
```

By placing the signal IO operations as close as possible to their corresponding wait statement, the HW intent is very clear. And there is no benefit either in terms of simulation performance or HLS QOR if they are placed further away from their corresponding wait statement.

Let's look at another similar example, which now also uses a Matchlib Connections blocking Pop operation:

```
 sc_in<int> i1;
 sc_in<bool> go;
 sc_out<int> o1;
 Connections::In<int> pop1;
 void my_thread() {
   int new_val=0;
   while (1) {
     o1.write(new_val);
     do {
       wait();
     } while (!go.read());
     int pop_val = pop1.Pop();
     new_val = i1.read();
     new_val = some_function(new_val + pop_val); // function has no internal IO
   }
}
```

Under the anchoring rules, i1.read() remains associated with the same synchronization point regardless of the intervening Pop. However, in raw pre-HLS simulation, a blocking Pop may stall for one or more cycles, so i1 can change before the i1.read() executes—creating behavior that differs from the anchored interpretation (and potentially from post-HLS scheduling). The fix is simply to place i1.read() immediately after its intended wait() anchor; this removes the mismatch risk without affecting QOR or simulation performance.

To automatically avoid all such potential pre-HLS versus post-HLS simulation mismatches, HLS tools may provide error or warning messages in cases where models have the pattern shown above. Precisely speaking: if a blocking message-passing operation separates a signal read or write operation from its corresponding synchronization interface call, then the HLS tool may emit an error or warning indicating that reordering the signal IO operation and the message-passing operation in the source text is advisable.

Another scenario in which RULE 1 applies is shown below:

```
 sc_in<bool> go;
 sc_out<int> o1;
 void my_thread() {

   while (1) {
     wait();              // WAIT 1
     o1.write(some_value);
     if (go.read()) {
       some_value = some_function();
     }
     else {
       wait();            // WAIT 2
     }
     some_other_function();
     wait();              // WAIT 3
```

```
   }
}
```

The signal read of `go` is clearly and uniquely associated with WAIT 1. However, the signal write of `o1` associates with WAIT 2 if `go` is false and WAIT 3 if it is not. This is a violation of RULE 1 and should be flagged as an error during HLS. The fix, as before, is to move the signal IO operation as close as possible to its intended wait statement so that the association is unconditional.

Next, let's consider *rolled* (or *preserved*) loops that perform signal IO within the loop body. Consider the following example:

```
sc_in<int> i1;
sc_out<int> o1;
void my_thread() {
    wait(); // reset state
    while (1) {
      wait(); // start of while loop
      #pragma hls_unroll no
      for (int i=0; i < 10; i++) {
        o1.write(i1.read() * i);
      }
  }
}
```

Note that the `i1.read()` operation is located inside the `for` loop, so presumably the user's intent is that it should be read as the loop iterates. *If that is not the user's intent, they simply should move the i1.read() operation before the loop start.*

In the post-HLS simulation, each iteration of the loop will consume at least one clock cycle, and a new value for i1 will be read (and a new value for o1 written) on each iteration. Again, this is the user's intent as per the code. In the pre-HLS simulation, the for loop body will execute in zero time, and only the last write to o1 will have any effect. The solution to avoid this mismatch is to manually place a `wait()` statement within the for loop body so that the signal IO synchronization is explicit in the pre-HLS simulation.

RULE 2: If you have signal IO operations within *rolled* (or *preserved)* loops, manually place a wait statement within the body of the loop to avoid pre-HLS versus post-HLS simulation mismatches, and while doing so also follow RULE 1.

To automatically prevent these types of pre-HLS versus post-HLS simulation mismatches, HLS tools may emit warning or error messages if they encounter a rolled loop which has signal IO operations within the loop body, and the loop body does not have a wait statement included within the loop body. (Cat4)

If a pre-HLS model adheres to RULE 1 and RULE 2, then all signal IO in the post-HLS model shall occur only at clock cycles that correspond to explicit wait statements or explicit synchronization statements. For direct-input signals/ports (e.g., #pragma hls_direct_input and #pragma hls_direct_input_sync), this requirement refers to the logical (Σ-visible) anchor cycle of the Read/Write: the event is still considered to occur at its synchronization point even if HLS physically samples a stable signal later; such late-sampling micro-steps are ε-steps and are not Σ-visible. User designs that do not adhere to both RULE 1 and RULE 2 are *ill-formed.*

Returning to the Analogy from RTL Synthesis

At the beginning of this document, we presented the example of a Verilog sequential block with an output coded like:

```
Out1 <= #some_delay new_val;
```

Recall that in Verilog simulation, if some_delay is less than the clock period of the block, then it will probably not affect the overall cycle level behavior of the system during simulation. However, if some_delay is more than the clock period, it probably will.

During RTL synthesis, all RTL synthesis tools will ignore all delays in the input model, in this case even if some_delay is greater than the clock period. Some RTL synthesis tools might give a warning for the code above like "Simulation and synthesis results are likely to mismatch because delay in model is greater than clock period."

HLS tools that adhere closely to the *conceptual model* presented in this document should automatically provide errors or warnings for violations of RULE 1 and RULE 2 as described in the section above. This is analogous to the error message that the RTL synthesis tool would provide in the example directly above.

However, HLS synthesis tools that do not follow the rules described in this document might adhere in these cases more closely to the pre-HLS SystemC simulation behavior. In this case such HLS tools might not provide any errors or warnings for violations of RULE 1 and RULE 2. This is analogous to an RTL synthesis tool being very smart (maybe even too smart) about synthesizing matching HW based on the actual value of some_delay in the example directly above.

Summary

At the beginning of this document, we said that the intent was to present "no surprises" to a DV engineer who is using a single testbench to verify both the pre-HLS and post-HLS models. The key aspects of the document which support this are:

- Three groups of IO operations are defined (message-passing, signal IO, and synchronization calls) and each is treated uniformly. These IO operations are easy for verification engineers to understand because they are already using them in their testbenches.

- The document specifically avoids complex constructs such as *protocol regions* used in some HLS tools.

- The document preserves the ability of the pre-HLS SystemC model to be *throughput accurate* by using a library such as Matchlib.

- Synchronization calls can affect the scheduling (anchoring) of signal IO operations, and synchronization calls can affect the scheduling of message-passing calls. In the conceptual anchoring semantics of this document, message-passing calls do not affect the anchor cycle of signal IO operations (and signal IO does not affect the legality/order of committed message transfers) except through explicit synchronization. Practically, in raw pre-HLS SystemC

simulation a blocking message operation may introduce additional cycles of waiting, so signal IO statements that are lexically separated from their intended anchor can observe different values; RULE 1 addresses this by recommending placement of signal IO adjacent to its anchor.

- HLS cannot by default reverse the order of message-passing calls, so it cannot introduce new deadlocks *via call-order reversal*. For pipelined loops (overlapped iterations) and other transformations that change the timing of request/commit under bounded capacities, deadlock preservation depends on the additional pipelined-loop discipline described in this document (automatic flush / WFG-inertness), not on the no-reverse rule alone.

- HLS pipelining is largely a *don't care* from the perspective of the verification engineer. If the design and the testbench are insensitive to changes in latency, and if external array accesses are not reordered or rearranged during loop pipelining, then the possible use of HLS pipelining will not affect verification, provided the pipelines flush automatically. Even if the design or testbench are sensitive to changes in latency, or if they are sensitive to reordering or rearranging of external memory accesses due to the use of HLS loop pipelining, then the behavior of the DUT will only change in expected (rather than unexpected) ways. (Appendix G provides formal guarantees regarding the equivalence of the pre-HLS and post-HLS systems.)

- Signal IO operations in the post-HLS model always occur exactly at synchronization points (e.g., wait statements) that are explicit in the pre-HLS source. (For direct-input signals/ports, "occur at synchronization points" is meant in the logical/anchoring sense: the Σ-visible Read/Write is timestamped at the synchronization point even if the implementation samples the stable signal later; those internal late samples are ε-steps.)

Appendix A – Factory Analogy

The scheduling rules described in this document apply to pre-HLS and post-HLS HW models. Although the rules may seem abstract and perhaps even arbitrary, they are shaped by the need to model systems in the real world that are required to have predictable behavior.

To understand the motivation behind the rules, it might be helpful to draw a simple real-world analogy and its correspondence to the rules described earlier.

Consider a factory that produces various types of wooden furniture:

The factory consists of people (processes) stationed at workbenches with various tools.

Each person is given written instructions about the specific tasks they are to perform (C++ code within a process).

People are instructed to send or receive objects (messages) to or from other people in the factory.

Sending or receiving objects may be blocking or non-blocking from the perspective of a person.

There is a clock with a second hand on the factory wall that everyone can see. (HW clock).

People have colored flags they can raise or lower to communicate with other people (signals).

If someone raises or lowers a flag, this is only seen by others the next time the clock second hand is at the top of the clock. (propagation delay of signals between sequential processes)

A person can choose to pipeline the tasks that they were assigned by hiring subordinates (pipeline stages) and having each one do a subtask. In general, this will improve the throughput for the tasks that the person was assigned.

It is possible for two people to explicitly synchronize their work by communicating via a synchronization protocol such as a barrier (synchronization).

It is possible to restart the work of some or all the people by raising a reset flag (HW resets).

Assume:

1. That the time that people take to complete their various tasks is in general variable, and similarly that the time that objects (messages) take to pass between people is in general variable.
2. That each person in general wants to complete their tasks as quickly and efficiently as possible.
3. That the factory needs to be able to make multiple types of furniture at the same time. For example, it might make chairs that need to be different colors or have different styles.

To reliably produce output, each person in the factory will need to adhere to rules like those described in this document.

Here's a sketch of a specific way the above example relates to the scheduling rules:

Person 1 sends chair seats and chair backs to Person 2. This is done with blocking operations, and there is no storage capacity between the people when the objects are sent. (This means that a Push operation cannot complete until the corresponding Pop operation is performed.) Assume that the fastest an object can be sent between people is 1 minute.

The written instructions that person 1 is given are:

```
while (1) {
 // internal processing code for seats and backs ...
 seats.Push(seat_object);
 backs.Push(back_object);
}
```

The written instructions that person 2 is given are:

```
while (1) {
 seat_object = seats.Pop();
 back_object = backs.Pop();
 // internal processing code for seats and backs ...
}
```

If both person 1 and person 2 choose to perform their tasks sequentially as written, then objects will be passed over time as:

```
            Person 1            Person 2
Minute 1:   seats.Push(seat1);
Minute 2:   backs.Push(back1);  seat1 = seats.Pop();
Minute 3:   seats.Push(seat2);  back1 = backs.Pop();
Minute 4:   backs.Push(back2);  seat2 = seats.Pop();
Minute 5:                       back2 = backs.Pop();
```

If both person 1 and person 2 choose to perform their IO operations in parallel, then objects will be passed over time as:

```
            Person 1                                Person 2
Minute 1:   seats.Push(seat1); backs.Push(back1);
Minute 2:   seats.Push(seat2); backs.Push(back2);   seat1 = seats.Pop(); back1 = backs.Pop();
Minute 3:                                           seat2 = seats.Pop(); back2 = backs.Pop();
```

If person 1 chooses to perform his IO operations in parallel, and person 2 chooses to perform his IO operations sequentially as written, then objects will be passed over time as:

```
            Person 1                                Person 2
Minute 1:   seats.Push(seat1); backs.Push(back1);
Minute 2:                                           seat1 = seats.Pop();
Minute 3:   seats.Push(seat2); backs.Push(back2);   back1 = backs.Pop();
Minute 4:                                           seat2 = seats.Pop();
Minute 5:                                           back2 = backs.Pop();
```

If person 1 chooses to perform his IO operations sequentially as written, but person 2 chooses to perform his IO operations sequentially in the reverse order as it was written, then objects will be passed over time as:

```
            Person 1            Person 2
Minute 1:   seats.Push(seat1);
Minute 2:   backs.Push(back1);  back1 = backs.Pop();
Minute 3:                       seat1 = seats.Pop();
```

In this case the person 1 seats.Push(seat1) operation at minute 1 will not complete at the start of minute 2. This means that the person 1 backs.Push(back1) operation will never start, and thus the Person 2 back1 backs.Pop() operation will never complete. So, the system will be in deadlock.

This example directly corresponds to the ordering rules related to message-passing operations in the *basic conceptual model* within this document, and in particular the specific rule that disallows reversing the order of message-passing operations.

Appendix B – Formal Presentation of Pipelined Loops

Modeling convention for pipelined-loop overlap.

The post-HLS pipelined-loop implementation may overlap iterations and may internally accept/stage up to the finite bound B(c_in) of input values for younger overlapped iterations before older iterations' outputs are produced. This internal accept/stage behavior is part of the back-annotated channel realization counted in B(c_in) between the Σ-visible endpoints of c_in (producer Push_{c_in} and

consumer Pop_{c_in}). These internal accept/stage movements are modeled as ε-steps. **Σ** records only the committed boundary transfers at the **Σ**-visible endpoints. In particular, the consumer-side boundary Pop_{c_in} is the loop-body Pop_{c_in} operation from the pre-HLS source when that Pop is included in the chosen Σ / Σ_DUT projection. The externally relevant effect of pipelining is therefore captured entirely by B(c_in) (via boundary backpressure/acceptance behavior and overlap timing at the Σ-visible boundary), without introducing any new auxiliary or duplicate Σ-visible boundary Pops: the Σ-visible consumer-side Pop_{c_in} commits are still the same loop-body Pop_{c_in} operations from the pre-HLS source, though they may commit at different times (and thus at different prefixes) than in an unpipelined execution. Once the pipeline reaches a quiescent blocked state under the automatic flush discipline below, it does not initiate new blocking request assertions for younger overlapped iterations.

Multi-input Pop points and coupler policies.

When a pipelined loop consumes from multiple input channels (e.g., c1,c2), back-annotation still assigns a separate finite bound B(ci) and cycle-boundary occupancy occ_ci(t) to each Σ-visible boundary channel ci. In elaborated realizations (Appendix Q), B(·), occ_·(t), and E4 are interpreted per explicit abstract channel (including any introduced staged/bundle/join channels). Any conjunctive/join dependency (e.g., "join not met") shall be represented only via such explicit internal staged/bundle/join boundary channels in the elaborated model (not as cross-channel disablement at the Σ-visible endpoints), so that for Appendix K the blocking point is that explicit boundary and the relevant Front_P(σ) members there are jointly channel-disabled ("ALL disabled ⇒ stall"). For the detailed elaboration patterns and examples, see Appendix Q.

Practical note (Σ_DUT projection).

The loop-body Pop_{c_in} in the pre-HLS source (i.e., the consumer-side Pop_{c_in} operation, conceptually after the in-flight capacity B(c_in)) may be exposed in Σ_DUT if desired, but is typically omitted since it is latency-insensitive, and it is often practically difficult to expose/collect in the post-HLS RTL. The liveness/deadlock reasoning still accounts for this blocking point without requiring it to be Σ_DUT-visible: Appendix K defines WFG edges using the ChannelEnabled/ChannelDisabled status of the corresponding boundary Pop_{c_in} (via E4/occ_c/B(c)), so no additional internal interface needs to be treated as a separate Σ-observable channel.

Automatic flush (drain-only blocking discipline).

Potential deadlocks are avoided by having the pipeline automatically flush.

Terminology (pending blocking frontier / Front_P(**σ**)). In an ε-quiescent cycle, consider the set of pending (i.e., issued but not yet committed) source-level blocking message-passing operations (Pop/Push) in the current interval between adjacent synchronization calls. By the Issue/Commit linkage (Appendix C) together with BoundaryReq's definition at the Σ-visible boundary (as given in "Common Formal Definitions for Appendices I-K"), each such pending blocking op has its endpoint-owned boundary request asserted (Push: vld, Pop: rdy) until it commits; purely combinational channel-side structure (including couplers) may affect only the complementary mirror signal (Push: rdy, Pop: vld) and thus whether the op commits in that cycle after ε-settling. Define Front_P(σ) to be the set of such pending blocking operations that are minimal with respect to source program order within that interval (if multiple minimal blocking ops are pending, they form a frontier). We refer to Front_P(σ) as the pending blocking frontier (or "minimal pending blocking frontier"). This Front_P(σ) notion is distinct from NextFront_P (Lemma S2 / Corollary J.2), which denotes the next Σ-visible candidate frontier and may include synchronization/signal-anchored events as well as message actions.

A pipelined loop is said to automatically flush if, whenever the pending blocking frontier Front_P(σ) is non-empty and every op ∈ Front_P(σ) cannot make progress because it is not channel-enabled (evaluated after combinational ready/valid/backpressure has ε-settled for the cycle), then:

1. Block point. The process is blocked on message passing at that pending blocking frontier Front_P(σ), as in the unpipelined source.
2. No new later work. While blocked at that frontier (i.e., while Front_P(σ) remains non-empty and all of its members are channel-disabled), the implementation does not issue any new blocking Pop/Push operations that are later in source program order (including any younger overlapped iterations) and therefore does not begin asserting their BoundaryReq.
3. No withdrawal. While blocked, the implementation does not withdraw any already-asserted blocking Pop/Push request (no "try-and-withdraw"); outstanding blocking requests remain asserted until they commit.
4. Clarification (Frontier-minimality vs. other asserted requests). The Wait-For Graph in Appendix K is intentionally defined from the minimal pending blocking frontier Front_P(σ), i.e., the current blocking causes. Accordingly, an already-asserted blocking request that is not in Front_P(σ) (for example, a request from a younger overlapped iteration that was issued before the blocked condition, or a downstream request draining work already past the frontier) may persist as protocol state under the "no withdrawal" rule, but it is not treated as a wait-for source while an earlier pending blocking op remains frontier-minimal. Such a request contributes WFG blocking only if/when it later becomes a member of Front_P(σ).
5. Drain-only downstream progress. Work that has already passed the blocked frontier point(s) may continue to drain, and any already-asserted resulting output-Push requests may commit when channel-enabled, provided no work advances past the blocked frontier point(s). (Formally: "no work advances past" is meant in the ≤_P / issue sense—while Front_P(σ) is fully disabled, P does not issue any new blocking message operation instance y such that op <_P y for any op in Front_P(σ); downstream drain may only complete/commit operations whose BoundaryReq was already asserted prior to reaching this blocked condition.)

(Equivalently: this is the "ALL disabled ⇒ stall" policy—P stalls only when none of the operations in Front_P(σ) are channel-enabled; downstream work may drain while Front_P(σ) is fully disabled.)

Example (II=1, latency=4, automatic flush): a loop that performs one blocking Pop_{c_in} and one blocking Push_{c_out} per iteration in the pre-HLS source may, after pipelining in the post-HLS RTL, internally accept/stage up to four input values into pipeline staging (pipeline overlap) before the first output Push_{c_out} commits, subject to the back-annotated bound B(c_in) (E4). Values may then reside for several cycles in internal pipeline storage; such internal staging/hand-off is ε-state within the back-annotated realization and does not create any new auxiliary or duplicate Σ-visible Push_c/Pop_c events beyond the boundary commits at the Σ-visible endpoints (it may change only the timing of those boundary commits under overlapped execution). The drain-only blocking behavior is as defined above under "automatic flush"; in particular, if the blocked operation is in a late pipeline stage (e.g., the final Push), there may be little or no downstream work to drain.

Appendix C – Formal Definitions of Issue and Commit

Scope / shared definitions. The definitions below are with respect to the Σ-visible endpoints of the chosen modeling boundary (Sys or Sys_B)—i.e., the ready/valid interfaces at which Σ records message-transfer events. All shared notions about (i) Σ vs. ε steps, (ii) BoundaryReq / ChannelEnabled / ChannelDisabled, (iii) non-blocking polls as ε on failure, and (iv) the B(c) / occ_c(t) (cycle-boundary, non-

fall-through) interpretation of E4 are as defined in “Common Formal Definitions for Appendices I-K”. Appendix C defines only Commit, Issue, and the Issue/Commit linkage conditions that make Issue(q) well-defined.

Definition: Commit (message channel) — as observed by an external clocked observer
Commit is the cycle in which an external clocked observer sees a transfer complete at a Σ-visible boundary endpoint: A message commits in the cycle when the observer sees both valid and ready high at the same time. The committed payload is the data value seen in that same cycle.

Definition: Issue (message channel) — as an auxiliary (scheduler/witness) timestamp
Issue(q) is an auxiliary timestamp used to state scheduling constraints (R4/E3/E5). Intuitively, it is the first cycle in which the calling process begins requesting the transfer corresponding to the source-level message operation instance q at the chosen Σ-visible boundary endpoint.
Issue/Commit linkage (well-formedness conditions). Fix a Σ-visible endpoint direction, and let req(t) denote the endpoint-owned boundary request signal in cycle t (valid for Push, ready for Pop), i.e., the process-driven request level captured by BoundaryReq at that boundary (per “Common Formal Definitions for Appendices I-K”). For each source-level message operation instance q on that endpoint direction:
• Boundary-request soundness (blocking requests are non-withdrawable). req(Issue(q)) = 1. If q is a blocking Push/Pop, then once issued it remains the unique outstanding request on that endpoint direction until it commits: req(t)=1 in every cycle t from Issue(q) through the commit cycle of q (inclusive), with stable payload while asserted for Push. (This is the “no deassert-before-commit” discipline used throughout the document.)
• Stable attribution while outstanding. If req(t)=1 and no transfer commits on that endpoint direction in cycle t, then any continued assertion req(t+1)=1 is attributed to the same outstanding q (i.e., attribution may not change from q to q′ without an intervening commit on that endpoint direction).
• Back-to-back issuance under continuously asserted req. If q_o commits in cycle t and the next source-level operation q_1 on the same endpoint direction is issued back-to-back, then Issue(q_1)=t+1 (even if the request signal does not deassert between commits), provided q_o and q_1 lie within the same source interval between adjacent synchronization calls.
• Sync-boundary discipline. If $q_o \in$ pref_S(s) and $q_1 \in$ suff_S(s) for a synchronization call s, then Issue(q_1) is strictly after s’s commit in the same trace (i.e., Issue(q_1) > clk_pre(s) in τ_pre and Issue(q_1) > clk_post(s) in τ_post). Any boundary request assertion that persists while the process is still pending at s is attributed to the outstanding operation(s) in pref_S(s) and does not constitute issuance of any operation in suff_S(s).

For non-blocking message passing (PushNB/PopNB), a failed poll produces no Σ-event and is modeled as an ε-step, while a successful poll contributes the corresponding committed Push_c(v) / Pop_c(v) event; see “Non-blocking message-passing” in “Common Formal Definitions for Appendices I-K” and Appendix I.2. The issuance attribution for any Σ-visible endpoint request signal—blocking or non-blocking—uses the linkage conditions above (stable attribution while outstanding).

Appendix D – Modeling Latency-Sensitive Protocols

In some cases, designs or testbenches may use protocols which are latency sensitive. These situations can be handled by isolating the latency-sensitive portions to small, self-contained parts, and then keeping the rest of the design and testbench latency insensitive.

Consider a case where a signal-level protocol is latency sensitive. The protocol will have specific timing behavior that it must meet. To handle this, we create a transactor that has two sides: the first side interacts with the signals and handles the detailed timing requirements, and the second side sends and receives messages with the rest of the system. The message-passing side is latency insensitive.

To properly model the detailed timing behavior, the transactor is modeled at the cycle-accurate level in SystemC. There is an example of such a transactor model in the following document and example:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/blob/master/matchlib_examples/examples/53_transactor_modeling/transactor_modeling.pdf

Appendix E – One-Way Handshake Protocols

In some systems a one-way signal handshake is sufficient for reliable system operation. When using a one-way handshake, the implicit assumption is that the "missing" handshake isn't required since it is always true.

For example, in time-domain signal processing hardware designs, signal processing HW blocks may input new samples at a fixed rate. The HW blocks are always ready to receive new samples on each clock, but they need to know if the samples are valid. These types of designs can be modeled with the one-way handshake dat/vld protocol demonstrated in this example:

https://github.com/Stuart-Swan/Matchlib-Examples-Kit-For-Accellera-Synthesis-WG/tree/master/matchlib_examples/examples/32_dat_vld

Appendix F – Scheduling Rules: Modeling Guidelines Summary

1) Prefer to use Connections::In/Out + SyncChannel over signal IO and wait().

2) Prefer to use Pop()/Push() over PopNB()/PushNB().

3) When pipelining a loop, prefer to use `hls_stall_mode flush`.

When using signal IO:

4) If modeling a cycle-accurate process, use `disable_spawn` and follow example 53_transactor_modeling style and do not use Push/Pop in the process.

5) If modeling a *direct input*, use `hls_direct_input` and possibly `hls_direct_input_sync`, and only change the signal at allowed times.

6) If combining signal IO with Connections::In/Out in same process, use proper signal IO handshake that does not rely on In/Out ports for process synchronization. Place signal IO operations very close to their wait() statements.

7) Unless you are modeling a cycle-accurate process, you should expect that latency will change between the pre-HLS and post-HLS models.

Appendix G – Equivalence Rules

This document informally states that a primary goal is to present "no surprises" to a DV engineer using the same testbench to verify the pre-HLS and post-HLS models.

Somewhat more formally, we can show how the scheduling rules within this document establish a set of equivalence rules between the pre-HLS and post-HLS models that provide strong guarantees about their behaviors.

The *basic conceptual model* establishes the base behavior for a single process:

1) Synchronization interface calls always remain in source code order.
2) Signal reads occur at closest preceding synchronization interface call.
3) Signal writes occur at closest succeeding synchronization interface call.
4) All message-passing operations before a call to a synchronization interface shall be issued before or in the same cycle in which the synchronization interface commits. (For blocking message-passing operations (Push/Pop), this implies the corresponding commits occur no later than the synchronization call's commit cycle.)
5) All message-passing operations after a call to a synchronization interface shall be issued after the cycle in which the synchronization interface call commits.
6) Two message-passing operations on separate interfaces which appear in sequence in the model may be issued either in the same sequence or in parallel in simulation and in synthesis, but they shall not be issued in the reverse sequence. Operations on the same message-passing interface are issued in source order (they are never reversed).
7) Synchronization calls can affect the scheduling of signal IO operations, and synchronization calls can affect the scheduling of message-passing calls, but message-passing calls cannot affect the scheduling of signal IO operations and vice-versa.
8) The STRICT_IO_SCHEDULING=relaxed option is not used.
9) We distinguish the rendezvous source model Sys (B(c)=0 for all channels) from the buffered source interpretation Sys_B, which uses bounded-FIFO channel capacities B(c) matching the post-HLS RTL (Appendices I–K).

Conceptually, systems are composed of many such processes which follow the basic conceptual model, and which communicate using only synchronization interface calls, latency-insensitive signal-level protocols, and committed message-passing transfers. If a design uses non-blocking message-passing calls (PushNB/PopNB), then unsuccessful polls are treated as internal ε-steps (no Σ-event), and each successful completion is represented in Σ as the corresponding committed Push_c/Pop_c transfer (Σ records transfers, not failed polls). If the resulting latency-sensitive behavior is externally visible at the DUT boundary, the snooping wrapper of Appendix L may be used to align the selected admissible pre-HLS execution with the RTL. Here we call this system of processes the *conceptual system*.

Practically, the modeling approach described above is too restrictive for real-world systems with optimized HW, so the scheduling rules in this document allow for key HW optimizations. But this is done

in a carefully controlled manner, such that the HW optimizations do not break equivalent behavior with the *conceptual system*.

Here is a summary of the HW optimizations that the scheduling rules allow, and a brief description of how we maintain equivalent behavior with the *conceptual system*:

1. Pipelined loops: In the post-HLS RTL, the pipeline may overlap iterations to improve throughput. See Appendix B for a formal presentation of pipelined loops.

2. Pipelined loops: HLS by default will not break any latency-insensitive signal IO protocols when pipelining loops.

3. Direct inputs: Signals that do not change after a process comes out of reset can be read as late as possible using *hls_direct_input*, saving register area.

4. Direct inputs: When *hls_direct_input_sync* is used, signals read by a process are updated at the precise point at which the HW pipeline is guaranteed to be ramped down, so the signal values do not need to be saved within pipeline registers, saving register area.

5. Memory accesses: HLS can reorder memory accesses within a process only if it can prove that the reordering is conflict-free.

6. Shared memories: Memories shared between processes are modeled as simple C arrays but always include explicit synchronization between processes in both the pre-HLS and post-HLS models.

7. Non-blocking message-passing interfaces: They are modeled at the level of committed transfers (successful completion), with unsuccessful polls treated as ε-steps and thus not part of Σ. If latency differences in these committed-transfer events are externally visible at the DUT boundary and must be aligned for verification, equivalent behavior between pre-HLS and post-HLS systems can be maintained by snooping the post-HLS system and using this to delay pre-HLS message arrival (Appendix L).

8. Latency-sensitive global signal IO: If latency differences are externally visible to the DUT, equivalent behavior between pre-HLS and post-HLS systems can be maintained by snooping the post-HLS system and using this to delay pre-HLS signals.

9. Latency-sensitive local signal IO: Isolate local latency-sensitive protocols to small, dedicated transactors, and use latency-insensitive signal IO or message-passing to communicate with the rest of the system.

10. One-way signal handshake protocols: Only use in cases where backpressure is not possible and embed assertions into the model to check that this is always true.

11. Combinational processes: If the pre-HLS model contains combinational processes, they will have identical behavior in the post-HLS model and thus they cannot introduce any differences between the models.

Taken as a whole, these rules allow full system verification to be performed on the pre-HLS system model. Full system verification does not need to be repeated on the post-HLS RTL system, provided the equivalence rules described above are followed. For liveness/deadlock guarantees (Appendices I–K), the results additionally assume weak fairness and channel-progress obligations for the scheduler/environment (Appendix N); the cycle-by-cycle R/E rules alone do not imply those progress properties.

Formalization of the Equivalence Rules in Appendix G

The following notation is used in this section.

Symbol Meaning

P	A process (thread or method) in the design
Σ	The alphabet of observable IO actions
τ_P	A (finite or infinite) per-cycle trace of observable actions executed by process P: for each global clock cycle t, τ_P[t] is the (possibly empty) set of Σ-actions committed by P at t. Actions committed in the same cycle are treated as simultaneous (unordered); only cross-cycle order (and the explicit E1–E5 constraints) is semantically significant.
S	The subset of Σ that are synchronization calls (explicit wait, SyncChannel, START_P/FINISH_P indicating process start/finish)
R	The subset of Σ that are signal reads
W	The subset of Σ that are signal writes
M	The subset of Σ that are committed message-passing transfers (i.e., committed Push_c / Pop_c events on message-passing channels, including successful non-blocking transfers). Unsuccessful non-blocking polls contribute no Σ-event and are modeled as ε-steps, and therefore are not members of M.
Q	The set of dynamic source-level message-operation instances executed by a process at the Σ-visible endpoints (blocking Push/Pop, and non-blocking PushNB/PopNB calls). Elements of Q may or may not commit.

A *system trace* is the tuple
τ = (τ_P)_P, one component per process.
For any action a ∈ Σ, let clk_pre(a) / clk_post(a) be the clock cycle at which a is committed.
For any issued message request q ∈ Q, issue_pre(q) / issue_post(q) denotes the auxiliary Issue timestamp for q in the pre-HLS / post-HLS trace, respectively, as defined in Appendix C and required to satisfy the associated Issue/Commit linkage (well-formedness) conditions with respect to the Σ-visible boundary request signal on q's endpoint direction.
If q commits, it contributes a unique committed transfer m(q) ∈ M with commit cycle clk_pre(m(q)) / clk_post(m(q)). If q does not commit (e.g., an unsuccessful non-blocking call), then m(q) is undefined / absent and no Σ-event is added to M.
(No deassert-before-commit assumption.) For blocking message requests q ∈ Q, once q is issued, its Σ-visible boundary request is not withdrawable: it remains asserted in every cycle from issue_pre(q) (resp. issue_post(q)) through q's commit cycle (if it commits), and for Push the payload remains stable while the request is outstanding.

1. Conceptual Model for a Single Process

For every process P the pre-HLS trace τ_P must satisfy the following *partial-order constraints*:

R1 (Program order of S).
∀ s1, s2 ∈ S : s1 <_src s2 ⇒ clk_pre(s1) < clk_pre(s2)

R2 (Location of R/W).
∀ r ∈ R : clk_pre(r) = clk_pre(pred_S(r))
∀ w ∈ W : clk_pre(w) = clk_pre(succ_S(w))
where pred_S(r) (resp. succ_S(w)) is the closest synchronization call that precedes (resp. follows) statement r in the source order. If no lexical predecessor (successor) exists, a virtual synchronization operation at time 0 (resp. ∞) is assumed.

R3 *(Isolation around S).*
Let pref_S(s) (resp. suff_S(s)) be the set of issued message requests q ∈ Q whose call is lexically before (resp. after) synchronization call s. Then
∀ q ∈ pref_S(s) : issue_pre(q) ≤ clk_pre(s) and ∀ q ∈ suff_S(s) : issue_pre(q) > clk_pre(s).
(For blocking message requests, if the synchronization call commits in the process trace, then any earlier blocking message requests in that source interval must have committed in some cycle ≤ that sync-commit cycle, by blocking-call control-flow semantics.)

R4 (Safe message issue ordering).
For every pair q1,q2 ∈ Q within the same process,
q1 <_src q2 ⇒ issue_pre(q1) ≤ issue_pre(q2).
(For same-channel operations, this simply states that calls on a single interface are issued in program order; for distinct channels, it forbids issuing them in reverse order.)

R5 (Channel capacity semantics; Sys vs. Sys_B).
Appendix G defines the rendezvous source model Sys, in which every channel has capacity 0. Concretely: for every channel c and any clock cycle t, the number of unmatched committed Push_c actions is zero and the number of unmatched committed Pop_c actions is zero; equivalently, no Push_c shall commit unless a matching Pop_c also commits at t, and no Pop_c shall commit unless a matching Push_c also commits at t.
In Appendices I–K we additionally use the buffered source interpretation Sys_B: the same source processes as Sys, but interpreted under bounded-FIFO channel semantics with capacities B(c) matching RTL_B (E4), with rendezvous recovered as the special case B(c)=0.

The source conceptual semantics for a process is thus a labeled partial order (Σ, ≤_P) generated by R1–R4 plus the channel semantics of Sys (rendezvous) or Sys_B (bounded FIFO), depending on which model is being referenced.

2. Equivalence Relation

Let τ_pre and τ_post be the Σ-traces (finite or infinite) of the same process before and after HLS, where each Σ-event is labeled exactly as in “Common Formal Definitions for Appendices I–K” (e.g., Push_c(v), Pop_c(v), Read(sig,val), Write(sig,val), and synchronization events).

Convention (S): all references to S, pred_S/succ_S, and <_src in E1–E5 use the source order of R2. Thus τ_pre denotes the Σ-trace of the pre-HLS source.

Σ-event matching convention. We use a canonical per-endpoint occurrence matching to make "the same Σ-events" precise. For each channel c, match committed Push_c and Pop_c events by their ordinal occurrence on that channel (e.g., the k-th committed Push on c in τ_pre matches the k-th committed Push on c in τ_post; likewise for Pop_c). For each such matched pair, the endpoint (c) and the payload/value label must agree. For the ordinal index k to be well-defined under the per-cycle "set/bucket" trace representation used in these appendices, we assume that for any fixed channel endpoint c, at most one committed Push_c and at most one committed Pop_c may occur in any single clock cycle.
For each observable signal sig, match Read(sig,·) and Write(sig,·) by their ordinal occurrence on that signal within the process trace, again requiring the value labels to agree. In forming τ_pre and τ_post, we apply the following canonicalization for observable signals: within each anchor interval between consecutive synchronization events in the order S, we record at most one Σ-visible Read(sig,val) and at most one Σ-visible Write(sig,val) for each (process, sig), with the anchor cycle given by E2. Any additional physical sampling of sig by the implementation, or redundant re-driving of sig within the same anchor interval, is treated as an internal ε-step and must not change the recorded value label. If a design intentionally relies on intra-interval changes of sig, or on distinguishing multiple reads/writes of sig within the same anchor interval, then sig must be modeled using an explicit synchronized interface (rather than as an ordinary observable signal) so that those distinctions become Σ-visible.
For synchronization events, match by their per-process occurrence index in the source order (R2). Independent Σ-events on different endpoints (including distinct message-passing interfaces) may commute and/or be grouped differently within a clock cycle as permitted by the R-rules and E3. Here "independent" is meant in the Σ-equivalence sense: the pair is not additionally ordered by any explicit Σ-visible ordering/timestamp constraint in E1–E5 (notably E3/E5); it is not a statement about (in)comparability under the Appendix G happens-before relation (→). Accordingly, τ_pre and τ_post are not required to be identical as a single total order, nor to agree on same-cycle "grouping" of distinct-interface Push/Pop operations, beyond the explicit ordering/timestamp constraints in E1–E5 below.

We say $\tau\_pre \approx \tau\_post$ iff the two traces match on Σ in this sense, and all of the following hold:

E1 (Synchronization-order preservation).
$\forall\ s1,s2 \in S : clk\_pre(s1) < clk\_pre(s2) \Rightarrow clk\_post(s1) < clk\_post(s2)$

E2 (Signal-visibility preservation).
$\forall\ r \in R : clk\_post(r) = clk\_post(pred\_S(r))$
$\forall\ w \in W : clk\_post(w) = clk\_post(succ\_S(w))$

E3 (Safe message issue ordering). (Post-HLS preservation of R4).
For any two message operations q1,q2 ∈ Q issued by the same process:
(i) Same-interface order is always preserved: if q1 and q2 access the same message-passing interface/channel and q1 <_src q2, then issue_post(q1) ≤ issue_post(q2).
(ii) Distinct-interface no-reverse: if q1 and q2 access distinct message-passing interfaces and appear in sequence in the source (q1 <_src q2), then issue_post(q1) ≤ issue_post(q2) (i.e., they shall not be issued in the reverse sequence).

(Note: any pipelined-loop internal staging/dequeue discussed in "Pipelined Loops" is ε-microstate within the back-annotated channel realization and is not a Σ-visible boundary issue event in Q nor a committed-transfer event in M; therefore it does not constitute a counterexample to (ii).)

E4 (Per-channel bounded-FIFO semantics).
For every observable channel c with capacity B(c), the projection of τ_post to events on c is FIFO-legal for that bound:
• No drop/duplicate + payload preservation: if the Push_c(v) events on c occur with payload sequence $v_1,v_2,\ldots$ then the Pop_c(v) events occur with payload sequence $v_1,v_2,\ldots$ (i.e., pops return the oldest unmatched pushed value).
• No overflow/underflow: for every cycle-aligned prefix π_t of the c-projection—i.e., π_t contains exactly the events on c committed in cycles < t—letting push(π_t) and pop(π_t) be the number of Push_c and Pop_c events in π_t, we have
$0 \le \text{push}(\pi\_t) - \text{pop}(\pi\_t) \le B(c)$.
Equivalently, the abstract occupancy after cycle t−1 is $\text{occ\_c}(t) = \text{push}(\pi\_t) - \text{pop}(\pi\_t)$.
Because π_t contains exactly the events committed in cycles < t, the availability predicates for commits in cycle t are evaluated against occ_c(t) at the start of cycle t (cycle-boundary, non-fall-through for B(c)>0): thus Pop_c may commit in cycle t only if occ_c(t) > 0, Push_c may commit in cycle t only if occ_c(t) < B(c), and simultaneous Push_c and Pop_c commits in the same cycle are permitted iff 0 < occ_c(t) < B(c).
Note: occ_c(t) is the logical occupancy of channel c at the chosen Σ boundary, induced solely by the Σ-visible boundary-commit history (Push_c/Pop_c) and the annotated capacity B(c) (i.e., the same "abstract FIFO state" referenced in Corollary J.2(2c)).

E5 (Messages cannot cross their immediate synchronization boundary).
For every synchronization call s (in the source order used by R3 / pref_S / suff_S):
$\forall\, q \in \text{pref\_S}(s) : \text{issue\_post}(q) \le \text{clk\_post}(s)$
$\forall\, q \in \text{suff\_S}(s) : \text{issue\_post}(q) > \text{clk\_post}(s)$
(For blocking message requests, if the synchronization call commits in the process trace, then any earlier blocking message requests in that source interval must have committed in some cycle ≤ that sync-commit cycle, by blocking-call control-flow semantics. Under the Issue/Commit linkage above, any boundary request assertion that persists while the process is still pending at s is attributed to the outstanding operation(s) in pref_S(s); no operation $q \in$ suff_S(s) may have Issue(q) ≤ clk_post(s), and any back-to-back issuance on that endpoint direction may occur only in the first cycle strictly after clk_post(s).)

See *Appendix I – Per Process Trace Equivalence Proof* for formal proof of the following:
For any single source-level process P that obeys the conceptual R rules and B rules, when interpreted as part of Sys_B (i.e., bounded-FIFO channel semantics with finite back-annotated effective capacities B(c) ≥ 0 matching the RTL system at the chosen Sys_B boundary—see Appendix J, Remark 2—hereafter called RTL_B), every finite observable trace produced by P has a matching post-HLS RTL trace that satisfies E1–E5. The rendezvous Sys case is recovered by setting B(c)=0.

---

3. Allowed Transformations and Why They Preserve ≈

Transformation permitted by the rules	Formal justification
Loop pipelining (overlaps iterations)	Introduces internal pipeline stages (modeled as ε-steps) and overlaps iterations. R1–R5 (and hence E1–E5) are applied to the explicit S set. Loop pipelining may commute independent Σ-actions across iterations (e.g., across distinct channels)—where "independent" here means "not additionally ordered by any explicit Σ-visible constraint in E1–E5 between the pair (notably E3/E5)," rather than "incomparable under Appendix G's happens-before relation (→)"—but it must still preserve: (i) per-channel FIFO legality (E4), (ii) the required issue discipline (E3) for source-ordered pairs, and (iii) the sync-boundary constraints (E1/E5) with respect to the (explicit + implicit) S-actions.
Overlap of internal dequeue/staging vs. later pipeline work (**ε**)	Permitted because the internal staging/acceptance activity is ε-internal within the back-annotated channel realization (as defined in "Appendix B - Pipelined Loops") and does not constitute a Σ-visible boundary message-issue reorder. Σ-visible boundary message issues must still satisfy E3, and per-channel committed-transfer legality is ensured by E4; automatic flush / Appendix K addresses deadlock preservation for overlap timing.
#pragma hls_direct_input (late sampling of static signals)	This pragma imposes a designer/environment stability contract: the signal's value must remain stable after reset (or after its last permitted update) while the receiving process may consume it. The logical signal Read actions r remain anchored exactly as in E2 (clk_post(r) = clk_post(pred_S(r))). HLS may implement this by sampling the signal later (instead of inserting internal storage), but because the signal is stable the sampled value equals the value at pred_S(r); therefore E2 and ≈ are preserved.
#pragma hls_direct_input_sync (controlled updates)	This pragma imposes an explicit timing contract on the environment: the controlled direct-input signals may change only on the cycle of the designated synchronization call s* (i.e., during the sync handshake). With that contract, the logical anchoring required by E2 is preserved: for the receiving process, the relevant Reads are anchored at the closest preceding synchronization call pred_S(r) = s*, and any environment update Write w_update is aligned with synchronization (succ_S(w_update) = s*). HLS may still implement late sampling internally, but the value observed is the same as the value at the E2 anchor point, so ≈ is preserved without modifying E2. Instrumentation note (logical anchoring). In the Σ-traces used for ≈ checking, each signal Read/Write event is timestamped at its E2 anchor point (e.g., clk_post(pred_S(r)) for Reads), not at any later physical RTL sample cycle that

Transformation permitted by the rules	Formal justification
	HLS may introduce. Any internal late-sampling micro-steps are treated as ε-steps. Therefore, equivalence checking must instrument/record the logical Read/Write events at the anchor (or use a wrapper/transactor that exposes only those anchored events), rather than naively timestamping the physical sample.
Memory-access reordering proven conflict-free	Let addr(a) be the symbolic address of access a. If the tool proves $addr(a_1) \neq addr(a_2)$ for the overlapped window, then the commutation of a_1 and a_2 is observationally silent; no external signal/message depends on the internal order.
Implicit FSM states / added latency	Extra states introduce *silent* ε-steps between observable actions; the partial order on Σ is unchanged, so E1–E5 are unaffected.
Shared-memory arrays with explicit synchronization	When a memory is accessed by multiple processes, the *designer* explicitly coordinates access with a sync operation modeled as an S-action. Because these S-actions appear in both traces, the read/write order is fixed by E1–E2, port-level FIFO legality is preserved by E4, and the overall equivalence relation ≈ still holds.
Non-blocking message-passing (PushNB/PopNB) verified by post-HLS snooping	If the latency differences are externally visible to the DUT, a verification wrapper delays the pre-HLS side so that the committed transfer events (successful completions) occur in the same order/timing as observed on the post-HLS channel. This wrapper is itself purely synchronising (S), so it cannot violate R1–R5. Once the wrapper is assumed, τ_pre and τ_post coincide on the committed message-transfer history recorded in Σ (i.e., the Push_c/Pop_c commit events), so E3–E5 are preserved. Unsuccessful non-blocking polls remain ε-steps and are not required to match. This is not a transformation that inherently preserves ≈, but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive *global* signal IO matched by snooping	If the latency differences are externally visible to the DUT, the same “mirror-and-delay” wrapper used for NB channels is applied to any globally-visible signal whose timing matters. Because the wrapper inserts only S-actions, the partial-order constraints are unchanged. This is not a transformation that inherently preserves ≈, but rather a verification technique to enforce equivalence for inherently latency-sensitive operations.
Latency-sensitive *local* signal IO isolated in cycle-accurate transactors	Local timing-exact protocols are confined to dedicated transactor processes that expose only latency-insensitive M or S operations to the rest of the design. Since the transactor boundary is now the observable interface, R1–R5 apply

Transformation permitted by the rules	Formal justification
	outside the timing-sensitive region, so system-level ≈ still holds.
One-way signal-handshake protocols with run-time assertions	For single-direction vld or rdy signals, an assertion proves that back-pressure can *never* occur. That assertion establishes a refinement in which the missing handshake is equivalent to a permanent logical '1', so the protocol is observationally identical to a two-way handshake that trivially satisfies E2.
Combinational processes	These have no S, R, W, or M actions, so their τ_P is the empty trace. The equivalence relation therefore holds vacuously.

For the purposes of the formal analysis, non-blocking message-passing is modeled at the level of committed transfers: a successful PushNB/PopNB contributes the same committed Push_c/Pop_c event to Σ as a blocking Push/Pop, while unsuccessful polls are ε-steps.

If a design contains latency-sensitive global signals or non-blocking transfers whose timing/ordering is externally visible at the DUT boundary and must be reproduced exactly in both simulations, we take as an axiom that a purely synchronizing snooping wrapper can be applied to supply the same latency choices to the pre-HLS simulation as those observed in the post-HLS RTL (Appendix L). Conceptually, this wrapper is a witness-selection device: it does not enlarge the set of admissible pre-HLS behaviors, but restricts the pre-HLS run to one admissible execution whose Σ-events align with the observed RTL execution. Under this wrapper, the committed Σ-traces of the two simulations agree, and the equivalence rules E1–E5 apply to the wrapped runs.

Appendix L provides detailed guidance to enable this perfect alignment to be achieved in practice.

---

4. Proof Structure

The proofs proceed in three stages:

First, in Appendix I we establish the per-process witness correspondence needed by the later system-level proof: under a live environment, every post-HLS RTL_B process trace (equivalently, every reachable finite observable prefix) has a matching source-level trace in Sys_B (Post → B), satisfying E1–E5. The converse direction (B → Post) is not claimed for all Sys_B traces in general; when it is needed, it is restricted to RTL-consistent prefixes (e.g., via Appendix L's snooping / witness-selection technique). Here, live environment does not mean assuming the entire system is deadlock-free; it refers specifically to the standard conditions of weak fairness, finite pipeline depth, and the local progress invariants (e.g., P_push(c),P_pop(c)). These are scheduling and channel-usage side-conditions, not the global liveness property that will be proved later.

Second, in Appendix J we compose these per-process results into a system-level equivalence theorem. This stage relies on channel-ordering and occupancy lemmas but does not assume system-level liveness.

Finally, Appendix K discharges the earlier "live environment" assumption by proving that system-level liveness is preserved through HLS. Specifically, if the source-level bounded-FIFO interpretation Sys_B (with capacities B(c) matching RTL_B) is live under weak fairness, then RTL_B is also live. (When B(c)=0 for all channels, Sys_B coincides with the rendezvous model.) This step ensures that

the overall proof is non-circular: the liveness property invoked in the first stage is rigorously established only at the end.

---

5. Causal Dependency Between Processes

To prove system-level equivalence, we must distinguish between incidental timing and functionally significant ordering.

Formal Definition of Causal Dependency

To formalize the notion of functionally significant ordering and resolve ambiguity, we define the happens-before relation, denoted by ->, on the set of all observable IO actions (Σ) across all processes in the system. The relation is causal (information-flow) rather than purely temporal: events related by → may still commit in the same clock cycle (e.g., rendezvous), but the direction of → indicates which event can be depended on by the other. This relation establishes a strict partial order of events, where a -> b is read as "a happens before b".

The *happens-before* relation is the smallest relation satisfying the conditions below:

1. Intra-Process Order: If actions a and b occur within the same process P and a precedes b in the program's execution trace, then a -> b. This directly reflects the sequential nature of the code within a single thread. Clarification (non-temporal). Here “precedes” refers to the order of IO action *instances* in the process’s trace/program order (i.e., the order induced by the process’s execution), not the relative order of their commit cycles; distinct-interface message actions may be issued in order while committing in either order depending on backpressure.

2. Inter-Process Communication: The relation is established by the direct exchange of information or synchronization between processes.

    - Message-passing (committed transfer): If a is the commit of a send on channel c in process P1 (a committed Push_c, including a successful PushNB), and b is the commit of the corresponding receive on c in process P2 (a committed Pop_c, including a successful PopNB), for the same logical message instance, then a → b. Here “corresponding receive” is defined by the channel’s FIFO semantics: b is the Pop_c commit that consumes the element produced by a (matching is by FIFO position / transfer instance, not by payload-value equality; duplicate payload values do not introduce ambiguity). Equivalently, if we enumerate committed Pushes on c as Push_c^1, Push_c^2, … in commit order and committed Pops as Pop_c^1, Pop_c^2, … in commit order, then Push_c^k → Pop_c^k. For rendezvous channels (i.e., B(c)=0), the matching Push_c and Pop_c commits occur in the same clock cycle; nevertheless we still include the directed edge a → b to reflect unidirectional message transfer (the Pop observes the value supplied by the Push) and to allow system-level causal chaining via transitivity. This edge does not assert an earlier commit cycle—only causal precedence—and no reverse edge (b → a) is implied unless there is an explicit reverse-direction communication (e.g., another channel or a signal handshake). Unsuccessful non-blocking polls are ε-steps and do not participate in →.

    - Signal Handshake: If a is a signal write action w(sig) in P1 and b is a signal read action r(sig) in P2 that reads the value written by a as part of an explicit handshake protocol,

then a -> b. A complete two-way handshake (e.g., vld/rdy) creates a causal chain, such as w(vld)@P1 -> r(vld)@P2 -> w(rdy)@P2 -> r(rdy)@P1.

- Explicit Synchronization (two-party SyncChannel / ac_sync): Let *sout* be the commit of a SyncChannel sync_out (or equivalent ac_sync call) in process P1, and let *sin* be the commit of the matching sync_in in process P2 for the same synchronization instance. Then this synchronization acts as a two-process barrier:
  - Every observable action in P1 that occurs before *sout* (in P1's trace/program order) satisfies a→b for every observable action b in P2 that occurs after *sin*.
  - Symmetrically, every observable action in P2 that occurs before *sin* satisfies a→b for every observable action b in P1 that occurs after *sout*.
  - (The two sync commits *sout* and *sin* are treated as simultaneous; we do not impose *sout*→*sin* or *sin*→*sout*.)

3. Transitivity: The relation is transitive. If a -> b and b -> c, then a -> c.

Using this relation, we now provide a formal definition for causal dependency between synchronization events, which are the fundamental ordering anchors in the scheduling model.

Definition: Causal Dependency
A synchronization event s2 in process P2 is causally dependent on a synchronization event s1 in process P1 if and only if s1 -> s2.
If neither s1 -> s2 nor s2 -> s1 holds true, the events s1 and s2 are considered causally independent (or concurrent). Any change in the observed temporal ordering of causally independent events between the pre-HLS and post-HLS simulations is considered an incidental, functionally insignificant variation in latency. A well-formed design, under this methodology, shall not rely on a specific ordering of causally independent events for functional correctness.

To be clear, the shall not rely requirement is a design rule. To adhere to this design rule, designs must use message passing, SyncChannel operations, and signal handshakes to properly order operations across the system. Designs which violate this design rule are outside of the formal guarantees.

---

6. System-Level Theorem

Theorem (Compositional Equivalence).
Fix any test stimulus (environment inputs) and initial state. Let $\tau_{B,system}$ be the resulting observable trace of the source-level bounded-FIFO system Sys_B under that stimulus, where Sys_B denotes the same source-level processes as Sys but interpreted under E4 as abstract bounded FIFOs of capacity B(c) (the same B(c) as in RTL_B, see Remark 2 in Appendix J); when B(c)=0 this coincides with rendezvous system Sys under that stimulus, and let $\tau_{post,system}$ be the resulting observable trace of RTL_B under the same stimulus. Assume every process P satisfies $\tau_{pre,P} \approx \tau_{post,P}$ by Appendix I, every channel c has finite capacity $B(c) \geq 0$, the progress invariants P_push(c) and P_pop(c) hold for every c, and the system satisfies weak fairness (WF). Then the aggregate traces are equivalent: $\tau_{B,system} \approx \tau_{post,system}$ under rules E1–E5 (given the R rules and B rules).
Proof sketch. (Full proof is in Appendix J – System-Level Trace Equivalence Proof)
Equivalence is defined per observable channel or signal.
By E1–E5 each process preserves:

1. ordering of synchronizing actions visible to other processes,
2. FIFO legality of every channel,

3. atomic association between signal IO and its bounding sync.
4. the *side-of-sync* predicate for all message actions (E5).

Because channels/signals are the only inter-process observables, the missing "closed-system" lift from per-process equivalence to system equivalence is exactly the composition (congruence) argument proved in Appendix J: under the stated progress invariants and WF, $\tau\_pre,P \approx \tau\_post,P$ for each P implies $\tau\_B,system \approx \tau\_post,system$. (We claim trace equivalence under ≈, not a stronger bisimulation property.)

---

7. Practical Implication ("No Surprises" Guarantee)

If a verification environment exercises only the alphabet Σ (signals, message ports, synchronization calls) and does not test internal latency, then any testbench that passes on the pre-HLS model is provably guaranteed to pass on the post-HLS RTL under the assumptions of Theorem (Compositional Equivalence) above (in particular: the scheduling rules plus the progress invariants and WF). See Appendix J for the full system-level argument.

Appendix G thus establishes that the scheduling rules create a *trace-equivalence* relation between the high-level model and the synthesized RTL: every legal compiler optimization is a morphism of the labeled partial-order structure defined by R1–R5, ensuring functional indistinguishability at all observable interfaces.

Appendix H – Possible Criticisms

This section highlights several potential challenges in the verification approach of Appendix G.

1. Designer-Managed Shared-Memory Synchronization

Appendix G requires that any memory shared between processes use explicit synchronization primitives inserted by the designer. The HLS tool does not perform static verification of those primitives. In practice, we mitigate this risk by encapsulating shared memories within parameterized library classes (for example, see examples 12_ping_pong_mem and 15_native_ram_fifo) that are pre-verified for correct memory access coordination in both the pre-HLS and post-HLS models. Typically such classes will be parameterized for aspects such as element type and memory size.

2. Latency-Sensitive Signal Alignment ("Snooping")

See Appendix L for detailed discussion on snooping.

3. By Default, Matchlib Connections Model a Skidbuffer inside In<> Ports

(Note that this overall document and specifically Appendix G are not intended to be strictly tied to Matchlib. Instead, this document is intended to be applicable to any pre-HLS model written in SystemC using signals, message-passing channels, and synchronization calls.)

Matchlib Connections supports throughput accurate modeling in the pre-HLS simulation. Currently the throughput accurate modeling mechanism relies on the Pre() and the Post() methods using a 1-place buffer to store transactions that are in flight on Connections::In<> ports. This means that by default In<> ports function like a skidbuffer. Connections::Out<> ports do not introduce any additional storage capacity into the pre-HLS simulation.

By default, Catapult adds skidbuffers on input ports into the post-HLS design, so the formal equivalence relationship described in this document still holds. In other words, the default pre-HLS and post-HLS designs both still model rendezvous semantics, since both explicitly include a structural instantiation of a skidbuffer on input ports. In effect, we define the rendezvous boundary *before* the skidbuffer.

It is possible to remove some or all the skidbuffers during Catapult HLS. Because the abstract rendezvous boundary is defined before the skidbuffer, this choice is orthogonal to the equivalence rules (R1–R5/E1–E5) as long as skidbuffer-internal behavior is not included in Σ. However, to keep the pre-HLS simulation's *port micro-architecture* aligned with the RTL configuration, the following compile flag must then be used in this case:

```
-DFORCE_AUTO_PORT=Connections::SYN_PORT
```

This will remove all the skidbuffers from the pre-HLS model. In this case, if some of the skidbuffers are still inserted during HLS, the capacity effects of those specific skidbuffers should be added into Sys_B so that the formal equivalence relationship still holds.

In this case the recommended methodology is to use both approaches:

- Use the default throughput accurate mode in Matchlib for most analysis and debug, and for pre-HLS performance verification.

- To keep the pre-HLS simulation's *port micro-architecture* strictly aligned with the RTL configuration, then use -DFORCE_AUTO_PORT=Connections::SYN_PORT, and rerun final verification tests.

Common Formal Definitions for Appendices I-K

Throughout Appendices I–K, the formal comparison is between RTL_B and Sys_B, i.e., the same source processes interpreted under bounded-FIFO semantics with capacities B(c) matching the RTL (E4). The rendezvous model is recovered as the special case B(c)=0.

Symbol / Rule	Definition — concise but complete
BoundaryReq(m, t)	For any Σ-visible message-passing operation $m \in$ {Push_c(v), Pop_c(v)} and cycle t, BoundaryReq(m,t) means: at the chosen observable boundary, the endpoint is requesting that transfer at cycle t. Operationally (ready/valid): • for Push_c(v): the producer endpoint asserts vld_c (with stable payload as required), and • for Pop_c(v): the consumer endpoint asserts rdy_c.

Symbol / Rule	Definition — concise but complete
	BoundaryReq is therefore a predicate about the endpoint's boundary-visible request assertion (Push: producer asserts vld with stable payload; Pop: consumer asserts rdy). BoundaryReq refers to the endpoint-owned request signal driven by the process (Push: vld; Pop: rdy). Purely combinational realization structure on the channel side may affect only the complementary mirror signal at that boundary (Push: rdy; Pop: vld) and thus the commit condition after ε-settling, but it does not change the endpoint request assertion captured by BoundaryReq(m,t).
ChannelEnabled(m, t)	ChannelEnabled(m,t) means: BoundaryReq(m,t) holds, and in the ε-quiescent (settled) ready/valid fixed point of cycle *t*, the channel-side commit-enabling condition for *m* holds at the chosen boundary (i.e., *m* is not blocked by the channel's availability/back-pressure conditions under the Sys_B / E4 interpretation). Concretely, for bounded-FIFO channels B(c)>0, the channel-side condition is exactly the E4 availability predicate at that boundary (space for Push / data for Pop); rendezvous is the special case B(c)=0.
ChannelDisabled(m, t)	ChannelDisabled(m,t) means: BoundaryReq(m,t) holds but ChannelEnabled(m,t) does not. Intuitively: the endpoint is requesting the transfer, but the transfer is blocked by channel-side conditions (e.g., empty/full or the rendezvous partner not simultaneously requesting), as evaluated at the ε-quiescent handshake fixed point used throughout the document's enablement tests (e.g., Appendix K). Consequently, at the chosen boundary the channel-side enablement test remains exactly E4: for any Σ-visible message operation *m* on an abstract channel *c*, if BoundaryReq(m,t) holds and *c*'s E4 availability predicate holds at cycle *t* (space/data or rendezvous participation, as applicable), then ChannelEnabled(m,t) holds in the ε-quiescent handshake fixed point. Any cross-channel dependency introduced by combinational couplers must be reflected only at explicit internal staged/bundle/join boundaries introduced by the elaboration (i.e., via BoundaryReq/ChannelEnabled on those explicit staged/bundle channels); at the chosen Σ-visible channel endpoints, BoundaryReq and ChannelEnabled/ChannelDisabled for channel *c* may not depend on other channels' state beyond the per-channel E4 predicate for *c*, and no extra hidden predicate may block an otherwise E4-enabled boundary transfer.
Σ	Set of observable actions for the equivalence check (committed IO events) at the chosen observable boundary (often the DUT IO boundary for functional equivalence, but optionally expanded—e.g., for Appendix K deadlock reasoning): • channel commit events Push_c(v) and Pop_c(v) on channels designated as observable, labeled with the transferred message value (or abstract message identity) v • synchronization events wait(), Sync, START_P, FINISH_P • single-cycle signal actions Write(sig, val) and Read(sig, val) on signals designated as observable, labeled with the value written/read.

Symbol / Rule	Definition — concise but complete
	Internal-only channels/signals may be excluded from Σ (treated as ε-microstate) when they are not part of the chosen observable boundary. If such channels/signals are later brought into the observable boundary (e.g., by snooping for debug or analysis), then they become Σ-events and must match under ≈. Deadlock-analysis requirement (Appendix K): for Appendix K, Σ is instantiated to include every message-passing channel whose **Σ**-visible endpoint transfers at the chosen Sys_B boundary (Push_c/Pop_c) can contribute a dependency edge in the Appendix K Wait-For Graph, either because it is on the declared observable boundary or because it is snooped. Internal micro-interfaces that lie within the back-annotated realization counted in B(c) (e.g., post-HLS RTL pipeline stage implementation) are not treated as separate channels for this requirement: they may be exposed/snooped for debug, but Appendix K's WFG edges are defined using the ChannelEnabled/ChannelDisabled status of the corresponding boundary Push_c/Pop_c (via E4/occ_c/B(c)), so such internal points can participate logically in the deadlock argument without being explicitly Σ-observable. Non-blocking message-passing (PushNB/PopNB): A non-blocking call that returns "not completed" produces no Σ-event and is modeled as an internal ε-step. When a non-blocking call succeeds (returns "completed"), that success is represented in Σ as the corresponding committed Push_c(v) or Pop_c(v) event on that channel, labeled with the same transferred value/message v (i.e., Σ records transfers and their payloads, not failed polls). A non-blocking call is considered issued in the cycle in which it executes (its Issue cycle). A call that returns "completed" commits in that same cycle and therefore yields a committed transfer m(q) ∈ M; a call that returns "not completed" produces no committed transfer event (m(q) undefined / absent) and may be treated as an ε-step with respect to Σ-visible message-transfer events.
START_P	First observable action emitted by process P after reset. Marks the moment P begins executing user code. A new START_P is emitted each time P exits reset.
FINISH_P	Last observable action of process P. If P executes an unbounded loop, FINISH_P never occurs, and the trace is treated as infinite.
ε-step	An *internal*, unobservable RTL state transition (pipeline advance, FSM micro-state, etc.).
B(c)	Back-annotated effective capacity of channel c at the chosen Σ-observable Sys_B boundary (the same boundary used by E4 and Appendix K). Finite, B(c) ≥ 0. It reflects the total bounded "composite buffer" storage/credit in the RTL realization between the producer's and consumer's Σ-visible endpoints (e.g., FIFO storage, skid/elastic buffering, and any HLS-inserted pipeline staging that can hold in-flight messages).

Symbol / Rule	Definition — concise but complete
	• B(c)=0 ⇒ rendezvous (capacity-zero) channel. • B(c)>0 ⇒ bounded FIFO (of that effective capacity). Point-to-point endpoint assumption (single-producer/single-consumer): For every message-passing channel c (including both buffered channels with B(c)>0 and rendezvous channels with B(c)=0), there is exactly one producer process that may execute Push_c on c, and exactly one consumer process that may execute Pop_c on c. Hence the complementary endpoint relevant to any blocked Push_c/Pop_c on c is unique. Modeling note (shared resources / arbitration): If an implementation conceptually has multiple producers and/or multiple consumers for a logical resource, that sharing must be modeled explicitly (e.g., an arbiter process plus per-client point-to-point channels), rather than as a single multi-endpoint "channel." This keeps WFG wait dependencies well-defined and unique per edge. Single-transfer-per-cycle endpoint constraint (scalar channel interface): For every message-passing channel c, in any clock cycle t, at most one Push_c may commit and at most one Pop_c may commit. Equivalently, the set of commits on a fixed channel in a single cycle contains ≤ 1 Push and ≤ 1 Pop (it may contain one of each in the same cycle). For rendezvous channels (B(c)=0), a committed transfer occurs only as a matching same-cycle Push_c(v) and Pop_c(v) pair. For buffered channels (B(c)>0), a same-cycle Push_c and Pop_c do not imply fall-through; the channel semantics are cycle-boundary, non-fall-through as stated in E4. Modeling note (multi-lane / burst transfers): If an implementation can transfer k>1 messages per cycle on a logical resource, model it as k parallel point-to-point channels (or as a single widened message transferred by one Push_c/Pop_c per cycle) so that the Σ-level event model continues to satisfy the scalar constraint above.
occ_c(t)	Abstract FIFO occupancy at the Sys_B channel boundary: number of items committed by Push_c but not yet committed by Pop_c at that boundary. Equivalently (E4), occ_c(t)=push(π_t)−pop(π_t), where π_t is the Σ-visible boundary-commit prefix strictly before cycle t (i.e., excluding any Push_c/Pop_c commits that occur in cycle t), so occ_c(t) is the abstract occupancy at the beginning of cycle t. For Pure-FIFO channels with B(c)>0, the FIFO availability predicates (not-full/not-empty) are evaluated against this begin-of-cycle occupancy; hence a Pop_c cannot commit in a cycle that begins empty solely because a Push_c also commits in that same cycle (no fall-through), and likewise a Push_c cannot commit in a cycle that begins full solely because a Pop_c also commits in that same cycle. Clarification (no "second Push/Pop"): movements of an item within the RTL channel realization (e.g., FIFO→staging/skid buffering, staging/skid buffering→pipeline-stage registers, pipeline-stage→pipeline-stage handoff) are internal ε-steps; they do not constitute additional Σ-visible committed Push_c/Pop_c events beyond the boundary events counted by push(π_t)/pop(π_t).

Symbol / Rule	Definition — concise but complete
	Note: occ_c(t) is an abstract analysis quantity derived from Σ-visible boundary commits; in both Sys_B and RTL_B, processes do not directly read occ_c(t) and instead proceed solely based on the per-channel ready/valid outcome of their own Push/Pop attempts.
P_push(c)	*Finite-progress invariant (producer):* interpret "continuously enabled" as a persistent (non-withdrawable; no deassert-before-commit) ready/valid request. • Buffered case (B(c) > 0): If the producer's Push_c request remains asserted continuously from some cycle t_o onward while the channel is full (occ_c = B(c))—with stable payload while asserted—and the complementary endpoint continuously requests the matching Pop_c (persistent request asserted; Pop_c boundary request remains true), then some cycle $t' \geq t_o$ commits a complementary Pop_c (i.e., the full condition is eventually discharged). • Rendezvous case (B(c) = 0): If the producer's Push_c request remains asserted continuously from some cycle t_o onward (with stable payload while asserted) and the complementary endpoint continuously requests the matching Pop_c (persistent request asserted; Pop_c boundary request remains true), then some cycle $t' \geq t_o$ commits the rendezvous transfer—i.e., a matching Push_c(v) and Pop_c(v) commit in the same cycle.
P_pop(c)	*Finite-progress invariant (consumer):* interpret "continuously enabled" as a persistent (non-withdrawable; no deassert-before-commit) ready/valid request. • Buffered case (B(c) > 0): If the consumer's Pop_c request remains asserted continuously from some cycle t_o onward while the channel is empty (occ_c = 0)—and the complementary endpoint continuously requests the matching Push_c (persistent request asserted with stable payload; Push_c boundary request remains true)—then some cycle $t' \geq t_o$ commits a complementary Push_c (i.e., the empty condition is eventually discharged). • Rendezvous case (B(c) = 0): If the consumer's Pop_c request remains asserted continuously from some cycle t_o onward and the complementary endpoint continuously requests the matching Push_c (persistent request asserted with stable payload; Push_c boundary request remains true), then some cycle $t' \geq t_o$ commits the rendezvous transfer—i.e., a matching Push_c(v) and Pop_c(v) commit in the same cycle.
Equivalence rules (E1–E5)	*E1 Synchronization-order preservation*: For each process P, the sequence of synchronization events executed by P (wait(), SyncChannel, START_P, FINISH_P) appears in the same source order in the pre-HLS and post-HLS traces. *E2 Signal visibility*: For each process P, each signal Read in P occurs at the closest preceding synchronization event in P, and each signal Write in P occurs at the closest succeeding synchronization event in P, in both the pre-HLS and post-HLS traces (per R2/E2) (except explicitly declared direct-input exceptions). Interpretation (logical timestamping). The equations in E2 define the logical timestamps of Read/Write Σ-events (the anchor points at which equivalence is

Symbol / Rule	Definition — concise but complete
	checked). An implementation may physically sample a Read later than pred_S(r) (or physically apply a Write earlier than succ_S(w)) provided the value is stable across the anchor interval and the trace emitted for equivalence records the Σ-event at the anchor point (see the Instrumentation note in Appendix G). Direct-input pragmas (e.g., #pragma hls_direct_input and #pragma hls_direct_input_sync) are not exceptions to E2: they impose stability/timing contracts that allow late physical sampling (or controlled updates) while the logical Read/Write Σ-events used for ≈ checking remain timestamped at the E2 anchor points (see Appendix G, Direct Inputs and Instrumentation note). *E3 (Safe message issue ordering). (Post-HLS preservation of R4).* For any two message operations q1,q2 ∈ Q issued by the same process: (i) Same-interface order is always preserved: if q1 and q2 access the same message-passing interface/channel and q1 <_src q2, then issue_post(q1) ≤ issue_post(q2). (ii) Distinct-interface no-reverse: if q1 and q2 access distinct message-passing interfaces and appear in sequence in the source (q1 <_src q2), then issue_post(q1) ≤ issue_post(q2) (i.e., they shall not be issued in the reverse sequence). (Note: any pipelined-loop internal staging/dequeue discussed in "Pipelined Loops" is ε-microstate within the back-annotated channel realization and is not a Σ-visible boundary issue event in Q nor a committed-transfer event in M; therefore it does not constitute a counterexample to (ii).) *E4 FIFO legality*: for each channel c, the post-HLS committed (Push_c(v), Pop_c(v)) history is a legal bounded-capacity execution consistent with Sys_B for that channel (capacity B(c)), and reduces to rendezvous consistency when B(c)=0. In particular, on each channel, the committed Pop_c(v) events return exactly the FIFO-ordered sequence of values v previously committed by Push_c(v), with no drops, duplications, or reordering. *E5 (Messages cannot cross their immediate synchronization boundary).* For every synchronization call s (in the source order used by R3 / pref_S / suff_S): ∀ q ∈ pref_S(s) : issue_post(q) ≤ clk_post(s) ∀ q ∈ suff_S(s) : issue_post(q) > clk_post(s)

Additional Semantic Assumptions (B-rules)

The following B rules are process assumptions used within the formal proofs. These B rules are in addition to the R rules presented in Appendix G.

ID	Basic-process property (informal statement)	Why it is needed / how it is used
B1 Deterministic Trace Property	For any fixed test stimulus and initial state, the sequence of observable actions emitted by the post-HLS design is unique, including the channel identity and payload/value of each	Ensures the per-process and system-level equivalence proofs (App. I & J) can match *one* post-HLS trace to *one* pre-HLS trace without

ID	Basic-process property (informal statement)	Why it is needed / how it is used
	committed Push_c(v)/Pop_c(v) and the value of each Write(sig,val)/Read(sig,val). Internal ε-steps may differ between runs, but the externally visible Σ-trace (with labels) cannot. Clarification: B1 is assumed of RTL_B (post-HLS) only. The source-level models Sys/Sys_B may admit multiple ε-/latency-different executions with the same Σ-projection. When a single concrete witness execution of Sys_B is required for simulation/debug, Appendix L's snooping wrapper may be used to select one admissible witness whose latency-sensitive events align with the unique RTL Σ-trace.	branching on scheduler nondeterminism.
B2 Weak Fairness of the Scheduler (WF)	If an action's enabling predicate remains continuously true from cycle *t* onward, the scheduler must eventually select that action (within a finite, but unspecified, number of cycles). Applies to Push, Pop, and Sync operations.	Required for all progress arguments, especially the liveness lemmas in Appendix K and the channel-progress corollaries.
B3 Channel Progress Invariants	For every channel *c* with capacity B(c): • P_push(c): If a process P is stalled at a blocking $Push_{(}c_{)}$ with its boundary request true and a persistent request asserted (payload stable, if applicable) while the channel is full ($occ_{(}c_{)} = B(c)$), and the complementary endpoint process continuously requests the matching $Pop_{(}c_{)}$ (persistent request asserted; $Pop_{(}c_{)}$ boundary request remains true), then the transfer on *c* must eventually complete (in particular: for B(c)>0, some $Pop_{(}c_{)}$ eventually commits, freeing space; for B(c)=0, the rendezvous completion eventually occurs). • P_pop(c): If a process P is stalled at a blocking $Pop_{(}c_{)}$ with its boundary request true and a persistent request asserted while the channel is empty ($occ_{(}c_{)} = 0$), and the complementary endpoint process continuously requests the matching $Push_{(}c_{)}$ (persistent request asserted with stable payload; $Push_{(}c_{)}$ boundary request remains true), then the transfer on *c* must eventually complete (in particular: for B(c)>0,	Explicitly assumed in Appendix J to rule out permanent back-pressure loops in which both endpoints continuously request a transfer but it never completes, which are logically independent of the cycle-by-cycle R-rules.

ID	Basic-process property (informal statement)	Why it is needed / how it is used
	some $Push_{(c)}$ eventually commits, providing data; for B(c)=0, the rendezvous completion eventually occurs).	
B4 Finite Stutter Bound	Every process P has a finite constant depth D_P such that whenever P is not externally stalled (i.e., P is advancing internal micro-state toward its next observable Σ-action, or its next Σ-action is locally enabled), P can execute at most D_P consecutive ε-steps before either (i) executing a Σ-action, or (ii) reaching a stable waiting state in which P has no further ε-step available until some external/peer/environment condition changes the enabling predicates.	This guarantees the ε-stutter closure needed to construct witness mappings in Appendix I and to bound internal micro-latency (pipeline/FSM bookkeeping) by ≤ D_P. Unbounded waiting due to back-pressure or missing peer stimulus is governed by WF/B2 and the progress obligations B3, not by B4.
B5 System Quiescence Closure	If, at some cycle t_o, no observable actions are enabled in any process, then within at most max_P D_P further cycles the system reaches a fixed point and executes no additional ε-steps.	Needed to finish the starvation-vs-deadlock analysis in Appendix K: ensures an all-disabled state cannot hide behind an infinite tail of unobservable activity.
B6 Time-Divergence / Idle-Stutter Convention	Even after the fixed point of B5 is reached (no further internal ε micro-steps are possible), the global clock continues to advance. We model each subsequent idle clock tick as a stutter step that produces no Σ-event and leaves the global state unchanged.	For prefix/trace matching in E1–E5 and for the Exec_post / Exec_B conventions in Appendix J, these idle ticks are treated as ε-stutter. Consequently, any reachable finite execution prefix/state (including a deadlock cutpoint) is extendable to an infinite execution under the same stimulus by appending idle-stutter ticks.

Appendix I – Per Process Trace Equivalence Proof

I.1 Overview

This appendix establishes a per-process witness property in the direction needed by Appendix J's execution-level correspondence: under a fixed initial state and fixed external stimulus/environment behavior, every reachable finite observable prefix of the post-HLS model RTL_B has a matching finite observable prefix in the source-level bounded-FIFO interpretation Sys_B, satisfying E1–E5.

Importantly, the converse direction ("every Sys_B trace has a matching RTL_B trace") is NOT claimed in general when latency-sensitive / non-blocking effects are made Σ-visible (e.g., by observing non-blocking poll outcomes or other timing-sensitive events via an expanded observation boundary). In such cases,

Sys_B may admit additional Σ-traces (for example, extra unsuccessful poll observations) that do not match the RTL_B Σ-trace under the same stimulus. When a reverse correspondence is required, Appendix L's snooping / witness-selection technique is used to restrict attention to RTL-consistent Sys_B executions (see Corollary J.1).

Rendezvous channels are the special case B(c)=0.

I.2 Formal Preconditions

- Observable alphabet Σ is as defined in *Common Formal Definitions for Appendices I–K*: it consists of the event schemas {Push_c, Pop_c, Sync, wait, START_P, FINISH_P, Write, Read} instantiated only for channels/signals designated observable; additionally, all message-passing channels that can participate in Appendix K deadlock reasoning are designated observable (possibly via snooping) and hence included in Σ.
- Non-blocking message-passing interpretation. A successful non-blocking PushNB/PopNB contributes the same observable event as the corresponding committed Push_c/Pop_c on that channel. An unsuccessful non-blocking call (returns "false" / "not completed") contributes no Σ-event and is modeled as an ε-step. Nevertheless, non-blocking polling may still assert a Σ-visible endpoint request; issuance is attributed using the Issue/Commit linkage conditions in the Issue definition above (stable attribution while outstanding): a continuously asserted endpoint request with no intervening commit is attributed to one issued request $q \in Q$ (with Issue(q) at the first asserted cycle, or at t+1 immediately following the previous commit if the request never deasserts), and only a poll that succeeds contributes the corresponding committed transfer $m(q) \in M$. If multiple poll checks occur while the endpoint request remains continuously asserted, they are ε-steps within the same q (no additional $q \in Q$).
- Silent step Any internal RTL microstate transition is written **ε**. In particular, unsuccessful non-blocking polls, arbitration bookkeeping, and pipeline advances are ε-steps.
- Finite-pipeline-depth premise (FPD) There exists a finite constant pipe_depth(P) = D_P < ∞ such that once P begins internal micro-progress toward its next observable Σ-action (or that next Σ-action is locally enabled), P executes at most D_P cycles of ε-steps before either committing a Σ-action or reaching a stable waiting state with no further ε-step available until external/peer conditions change. In particular, a process cannot perform an unbounded number of ε-only failed non-blocking polls while making no progress toward any Σ-action; designs that can spin indefinitely without reaching either a Σ-action or a stable wait state violate FPD/B4 and are outside the model.
- Weak-fairness premise (WF) If a micro-operation remains continuously enabled, the scheduler eventually issues it. This is an assumption on the execution/scheduling environment (See Appendix N), not an automatic guarantee of "clock-synchronous RTL." In practice it requires that any arbiters/back-pressure logic that can affect whether an enabled operation is selected must be fair in the sense of B2/B3.
- Finite-progress invariants (B3) For every channel c, assume producer and consumer obligations P_push(c) and P_pop(c) hold, guaranteeing that data (or space) eventually becomes available. This is an assumption on the execution/scheduling environment (e.g., fairness of arbiters/back-pressure; see Appendix N), not a consequence of the cycle-by-cycle R-rules.
- Pure-FIFO channel semantics. For any message-passing channel *c* with *B(c) > 0*: if a process has issued a persistent request to transfer on *c* (i.e., the request remains asserted and the payload is stable, if applicable), then the channel-side commit-enabling predicate is exactly the FIFO availability predicate—namely:

- for Push_c: *occ_c < B(c)* (“not full”), and
- for Pop_c: *occ_c > 0* (“not empty”).
- Modeling note (fall-through / simultaneous empty/full cases). This abstraction is cycle-boundary, non-fall-through for B(c)>0: however, it does permit a Push and a Pop to both commit in the same cycle when the FIFO begins the cycle neither empty nor full ($0 < occ\_c(t) < B(c)$); in that case both commits are legal and the occupancy is unchanged. It does not permit a Pop to commit in a cycle that begins empty solely because a Push also commits in that same cycle, nor does it permit a Push to commit in a cycle that begins full solely because a Pop also commits in that same cycle. If an implementation uses a fall-through FIFO (allowing same-cycle pass-through when empty), model that behavior explicitly (e.g., as an explicit bypass/rendezvous path composed with the bounded FIFO), so that the Σ-level committed-transfer trace matches the intended cycle semantics.
- Operational interpretation (ready/valid): occ_c(t) is an abstract quantity derived from Σ-visible boundary commits; processes do not read occ_c(t) directly. In both Sys_B and RTL_B, whether a pending blocking Push/Pop can proceed is realized operationally by the settled per-cycle ready/valid handshake outcome (evaluated in the ε-quiescent fixed point). A concrete realization is correct iff, for a persistent request with BoundaryReq true, the ready/valid outcome agrees with the corresponding availability predicate above at the chosen boundary.

No additional arbitration/grant/back-pressure gating is part of the channel semantics; any cross-channel coordination in the realization must be expressed only through the settled ready/valid handshake behavior that determines ChannelEnabled/ChannelDisabled, without adding enablement conditions beyond E4. Process-local control (BoundaryReq) determines when an endpoint requests; realization logic determines whether that request is enabled (ChannelEnabled) in that cycle.

Therefore, when the boundary request is true and the relevant FIFO availability predicate holds continuously, the transfer is continuously enabled; by weak fairness (B2) the corresponding commit occurs after a finite (though not necessarily bounded) delay.

- No ill-formed signal IO Every signal read/write has a unique bounding synchronization operation in the source program; otherwise, the design is ill-formed. (From RULE 1 and RULE 2).
- Fixed-stimulus interpretation (reactive environments). If the “same stimulus” includes a reactive testbench and/or peer processes, it is interpreted as the same global Σ-causal behavior (same externally supplied inputs, and the same peer/environment responses to the same prior Σ-visible history), not as a function of P-local history alone. In Appendix I this interpretation is used only as a witness-replay condition: when a next RTL_B event e is already fixed in the chosen post trace prefix, any peer/environment enabling needed for e is part of that same fixed global Σ-causal behavior and is therefore replayed consistently for the matching Sys_B construction. This is not a standalone assumption that arbitrary Sys_B continuations are globally live.
- Non-blocking message passing interpretation. In the post-HLS model, a non-blocking call (PopNB / PushNB) is a poll: if it fails (no data / no space / rendezvous partner not ready), it produces no observable Σ action and corresponds to an ε-step at the per-process trace level. If it succeeds, it produces the corresponding committed Pop_c / Push_c Σ-event. In the source-level bounded-FIFO model Sys_B, we treat non-blocking polls analogously: an unsuccessful poll contributes only ε, and the first successful completion is the Σ-visible committed transfer.
- A non-blocking poll may still assert a Σ-visible endpoint request; we attribute issuance using the Issue/Commit linkage conditions in the Issue definition above (stable attribution while

outstanding), so that a continuously asserted endpoint request with no intervening commit is attributed to one issued request $q \in Q$ (with Issue(q) at the first asserted cycle, or at t+1 immediately following the previous commit if the request never deasserts), and only a poll that succeeds yields the corresponding committed transfer $m(q) \in M$. Repeated poll checks under a continuously asserted request are ε-steps within that same q.

- WC (Witness-compatible non-blocking outcomes). Because failed polls are modeled as ε (unobservable), all correspondence results in Appendices I–K are stated for witness-compatible comparisons: any ε-modeled non-blocking poll outcome (and any other ε-modeled internal choice that can affect the next Σ-visible control flow / next Σ-visible blocking frontier) is fixed/selected to be consistent with the compared RTL_B execution prefix τ_post (e.g., via the Appendix L witness-selection/snooping wrapper).
- Remark (NB-CF as a sufficient condition). If a design satisfies the following non-blocking control-flow property—failed PopNB/PushNB polls do not influence the next Σ-visible control flow/frontier—then the witness is trivial and WC holds automatically. Equivalently: between two consecutive Σ-events of a process, inserting/removing any finite sequence of failed PopNB/PushNB polls may change only internal ε-behavior/timing, not which Σ-visible action (Push/Pop/synchronization anchor) is next in program order.

I.3 Auxiliary Lemmas

Lemma I.0 (Message-issue discipline in RTL).
The RTL satisfies E3: (i) per-interface issue order is preserved for all message operations; and (ii) for distinct interfaces, the no-reverse rule holds.

Lemma I.1 (Bounded **ε**-chain to **Σ**-action or stable wait). Starting from any state of P, within at most D_P clock cycles P either (i) commits its next observable Σ-action, or (ii) reaches a stable waiting state with no further ε-step available until some external/peer condition changes the enabling predicates.

Proof. Direct from FPD/B4. ■

Lemma I.2 (Eventual space / data). Assume P is blocked on • Push_c with $occ_c = B(c)$, and the complementary endpoint continuously requests the matching Pop_c (persistent request asserted; Pop_c boundary request remains true), or • Pop_c with $occ_c = 0$, and the complementary endpoint continuously requests the matching Push_c (persistent request asserted with stable payload; Push_c boundary request remains true). Then a complementary Pop_c (respectively Push_c) commits within finite time.

Proof. (By B3 / Appendix N.) In either case, P is stalled at a blocking channel call while its persistent request for that call remains asserted (payload stable, if applicable). By the additional premise, the complementary endpoint also continuously requests the matching operation with its boundary request remaining true. Therefore the premises of P_push/P_pop (Appendix N) hold, and the corresponding transfer must eventually complete: for Push_c at full, some complementary Pop_c eventually commits (freeing space); for Pop_c at empty, some complementary Push_c eventually commits (providing data). ■

Lemma I.DR (Deterministic replay / source-state determinacy under fixed stimulus + witness). Fix a process P, a fixed external stimulus, and a fixed witness (as in WC). Consider two executions of the same P source code that start from the same initial source-level state and are run with that same witness. If they produce the same P-local Σ-label sequence, then for every k, the source-level state of P at the ε-quiescent cutpoint after the first k P-local Σ-events is identical in the two executions.

Proof. With fixed stimulus and a fixed witness (WC), the source-level transition from one ε-quiescent cutpoint to the next is deterministic as a function of the current source-level state and the next P-local Σ-label (including any Pop return value or anchored Read value). Induction on k.

I.4 Inductive Construction for Finite Traces (Post → B)

Let

$$\tau\_post = \tau\_post[0..n-1] \circ e \quad (\text{where } |\tau\_post| = n + 1)$$

be the next postHLS prefix of RTL_B for process P (under the fixed initial state and fixed stimulus). Assume by induction that we already have a matching Sys_B prefix $\tau\_B[0..n-1]$ satisfying E1–E5 with $\tau\_post[0..n-1]$. We extend τ_B by a finite ε-chain (written ε^*) followed by an observable action e' so that

$$\tau\_B \circ \varepsilon^* \circ e' \quad \text{matches} \quad \tau\_post.$$

Non-blocking note. For a non-blocking PopNB/PushNB call, unsuccessful polls are treated as ε-steps. This is sound for Post→B construction under the witness-compatible (WC) premise (I.2): ε-modeled poll outcomes are either (a) irrelevant to the next Σ-visible control flow/frontier (NB-CF holds), or (b) fixed/selected consistently with τ_post (e.g., via Appendix L), so they do not introduce an unmatched Σ-visible control-flow/frontier divergence.

Lemma I.3 (Finite **ε***-extension step for I.4).

Let $\tau\_post = \tau\_post[0..n-1] \circ e$ (where $|\tau\_post| = n + 1$) be the next postHLS prefix of RTL_B for process P (under the fixed initial state and fixed stimulus). Assume by induction that we already have a matching Sys_B prefix $\tau\_B[0..n-1]$ satisfying E1–E5 with $\tau\_post[0..n-1]$. Then there exists a Sys_B extension by a finite ε-chain ε^* followed by an observable action e' such that:

(i) $\tau\_B \circ \varepsilon^* \circ e'$ matches τ_post under the document's Σ-event matching convention; and

(ii) the extended prefix continues to satisfy E1–E5 (in particular, E3's message-operation issue-order clause holds via Lemma I.0).

Proof. By cases on the kind of Σ-event e. Lemma I.1 bounds P-local internal progress to the relevant call site by ≤ D_P ε-steps (or reaches a stable wait with no further ε-step available until external enabling changes). Once the matching operation is continuously enabled, weak fairness (B2) guarantees it is selected after a further finite (though not necessarily bounded) delay. For blocking message transfers, if temporary disablement is due only to bounded-capacity/full-or-empty conditions ($occ\_c = B(c)$ for Push_c, $occ\_c = 0$ for Pop_c) or $B(c)=0$ rendezvous-style enablement, Lemma I.2 guarantees eventual space/data (given persistent complementary requests), after which the transfer is continuously enabled and the B2 argument applies. Hence ε^* is finite in all cases. ■

Case note (what differs across event kinds). For Sync/wait and anchored signal I/O events, the finite ε^* step is a witness-replay argument, not an independent global-liveness claim. The lemma conditions on the already chosen next RTL_B event e and, via the fixed-stimulus interpretation above, replays the same peer/environment enabling from the same global Σ-causal history in the Sys_B witness construction. Thus Lemma I.3 does not assume that peer/environment progress follows from P-local history alone, and it does not preempt the system-level liveness/deadlock arguments of Appendices J and K. For message-passing events, the only internal source of delay is local channel availability (eventual space/data, or rendezvous counterpart), which is exactly Lemma I.2.

I.5 Extension to Infinite Traces (Post → B)

Fix a process P and a fixed external stimulus/initial state. (For reactive environments, interpret "fixed stimulus" per Appendix I.2, "Fixed-stimulus interpretation (reactive environments).")
Let ρ_post be any RTL_B execution under this stimulus that is within the intended Appendix J / Corollary J.1 scope (i.e., an execution whose finite prefixes lie in Exec_post; equivalently: an execution that is a fair/progress-satisfying infinite run, using the idle-stutter/time-divergence convention B6 when no further Σ-progress is enabled). Let τ_post,P be the Σ-observable trace of P induced by ρ_post.
Finite-**Σ**-event case. If τ_post,P contains only finitely many Σ-events $e_1 \dots e_n$ (i.e., after e_n no further Σ-event of P occurs), apply the finite-prefix construction in I.4 to obtain a Sys_B prefix τ_B,P[0..n) whose Σ-projection matches $e_1 \dots e_n$ and whose correspondence satisfies E1–E5. Then, by B6, extend τ_B,P[0..n) to a countably infinite Sys_B trace by appending idle clock ticks that produce no further Σ-events. This yields an infinite τ_B,P that remains Σ-equivalent to τ_post,P (viewed under the same B6 idle-stutter extension), since both sides have exactly the same finite Σ-prefix and no further Σ-events thereafter.
Countably-infinite-**Σ**-event case. Otherwise, τ_post,P has countably many Σ-events. We construct an infinite Sys_B trace τ_B,P by an explicit inductive limit of the finite-prefix construction in I.4.
Let τ_B,P[0..0) be the empty prefix. For each n ≥ 0, assume we have constructed a finite Sys_B prefix τ_B,P[0..n) such that its Σ-projection matches the first n observable Σ-events of τ_post,P and the correspondence satisfies E1–E5. Let e_n be the (n+1)-st Σ-event of τ_post,P. Apply the I.4 construction step to extend τ_B,P[0..n) by a finite ε* segment followed by a matching observable event e′_n, obtaining τ_B,P[0..n+1) while preserving E1–E5.
Because each extension step adds a finite segment, the increasing chain of prefixes defines a (countably infinite) Sys_B trace τ_B,P whose every finite prefix matches the corresponding prefix of τ_post,P. Therefore, τ_B,P ≈ τ_post,P (for countably infinite traces as well). Scope clarification: this inductive-limit step establishes the per-process Σ-trace witness only (prefix/trace matching under E1–E5); it does not by itself discharge global fairness/progress obligations (B2/B3) or prove that an arbitrary global Sys_B continuation is fair/progress-satisfying. Those execution-level existence/scope obligations are handled in Appendix J / Corollary J.1 via the Exec_post / Exec_B definitions and qualifiers. ■

Appendix J – System-Level Trace Equivalence Proof

Theorem J.1 (Compositional Equivalence)

Fix any test stimulus (environment inputs) and initial state. Let τ_B,system be an observable trace of the source-level bounded-FIFO system Sys_B under that stimulus, and let τ_post,system be an observable trace of RTL_B under the same stimulus. (If B1 holds, RTL_B is Σ-deterministic for the given stimulus/initial state.) Assume every process P satisfies τ_B,P ≈ τ_post,P by Appendix I, where τ_B,P is the projection of the Sys_B system trace τ_B,system onto P, every channel c has finite capacity B(c) ≥ 0, the progress invariants P_push(c) and P_pop(c) hold for every c, and the system satisfies weak fairness (WF). Assume also B1 (Deterministic Trace Property) when uniqueness of the trace is relied upon. Then the aggregate traces are equivalent: τ_B,system ≈ τ_post,system under rules E1–E5 (given the R rules and B rules).
Notation: Let <_src denote the textual order of operations within a single process in the source code.
Proof
We first establish key invariants, then prove each equivalence property.
Lemma J.1 (Channel Occupancy Invariant)
For every channel c and at every clock cycle t in the post-HLS execution, the occupancy occ_post(c,t) satisfies 0 ≤ occ_post(c,t) ≤ B(c).
Proof of Lemma J.1:

We argue by cases on B(c).
If B(c) > 0 (buffered FIFO): By definition, the abstract boundary occupancy occ_post(c,t) is derived from Σ-visible boundary commits (occ_post(c,t)=push(π_t)−pop(π_t) at the beginning of cycle t, i.e., excluding commits that occur in cycle t), and the buffered-FIFO availability predicates are evaluated against this begin-of-cycle occupancy (no fall-through). Therefore, in any cycle t, a Push_c can commit only if occ_post(c,t) < B(c), and a Pop_c can commit only if occ_post(c,t) > 0. Since each committed Push_c increases occ_post by 1 and each committed Pop_c decreases occ_post by 1 (and at most one of each may commit per cycle), a straightforward induction on cycles yields 0 ≤ occ_post(c,t) ≤ B(c) for all t.
If B(c) = 0 (rendezvous): a transfer can commit only when the complementary endpoint is enabled in the same cycle (Push_c and Pop_c commit together), so the cycle-aligned occupancy is always 0 and thus satisfies 0 ≤ occ_c(t) ≤ B(c)=0 at every cycle.
Lemma J.2 (Per-Channel Push/Pop Precedence)
Fix a channel c. Let Push_c[k] denote the k-th committed Push on c in the post-HLS execution, and Pop_c[k] denote the k-th committed Pop on c (k ≥ 1). By A9 (Single-transfer-per-cycle endpoint constraint), at most one Push_c and at most one Pop_c can commit per cycle on a fixed channel, so these "k-th" events are unambiguous. Then: clk_post(Push_c[k]) ≤ clk_post(Pop_c[k]).
Proof of Lemma J.2:

- For B(c)=0 (rendezvous), Push_c[k] and Pop_c[k] commit simultaneously, so equality holds.
- For B(c)>0, a Pop can only commit when the FIFO is non-empty. After i committed Pushes and j committed Pops, occupancy is i–j. For Pop_c[k] to commit, immediately before it commits we must have i–j > 0, hence i ≥ j+1 = k. Therefore the k-th Push has already committed by that time, i.e., clk_post(Push_c[k]) ≤ clk_post(Pop_c[k]). □

We now prove each equivalence property:
E1 (Synchronization-order preservation):
Fix any process P. By Appendix I, τ_B,P ≈ τ_post,P, so the synchronization events of P occur in the same source order in both traces. Since this holds for every P, E1 holds for the system traces τ_B,system and τ_post,system.
E2 (Signal-visibility preservation):
Fix any process P. By Appendix I (and rule R2 as used there), every signal Read in P is anchored to P's closest preceding synchronization event, and every signal Write in P is anchored to P's closest succeeding synchronization event, in both τ_B,P and τ_post,P. Since this holds for every P, E2 holds for τ_B,system and τ_post,system.
E3 (Safe message issue ordering):
Fix any process P. We must show that P's post-HLS execution satisfies E3 as stated in Appendix G: (i) for same-interface pairs q1,q2 ∈ Q with q1 <_src q2, we have issue_post(q1) ≤ issue_post(q2); and (ii) for distinct interfaces that appear in sequence in the source (q1 <_src q2), the operations are not issued in reverse order. This is exactly the per-process property established in Appendix I (Lemma I.0). Since it holds for every P, E3 holds for the system traces τ_B,system and τ_post,system.
E4 (Per-channel FIFO semantics):
For each channel c, we must show:

1. The sequence of Push_c and Pop_c operations in τ_post forms a legal FIFO schedule
2. No messages are dropped or duplicated
3. FIFO order is preserved in the Σ-trace sense: if a committed transfer m1 precedes m2 in the per-channel (Push_c/Pop_c) history required by Sys_B/E4, then the corresponding committed transfers appear in the same order in τ_post on that channel.

From Appendix I, each process preserves the order of its operations on each channel. Lemma J.1 establishes that $0 \le occ\_post(c,t) \le B(c)$ at all times, and Lemma J.2 establishes that, for each channel c, the k-th Pop cannot commit before the k-th Push. Together these imply:

- Pops never underflow and Pushes never overflow the bounded FIFO (legality).
- Each committed Pop consumes exactly one previously committed Push, in FIFO order (no drops/duplication; FIFO order preserved).
- Capacity constraints are respected at every clock cycle.

Therefore, the post-HLS (Push_c, Pop_c) history on each channel is a legal bounded-FIFO execution consistent with the source-level FIFO semantics required by E4.

E5 (Messages cannot cross syncs):

For every synchronization call s (in the source order used by R3 / pref_S / suff_S):

$$\forall q \in pref\_S(s) : issue\_post(q) \le clk\_post(s)$$
$$\forall q \in suff\_S(s) : issue\_post(q) > clk\_post(s)$$

(Note: For blocking message transfers, the "no-withdraw"/persistence semantics imply that if a blocking operation is issued in the pre-side interval before s, then it must also commit no later than s (and similarly on the post side), because the process cannot complete s until all earlier-issued blocking requests in that interval have committed.)

This property is guaranteed per-process by Appendix I and requires no inter-process reasoning, so it lifts directly to the system level.

Conclusion:

All five equivalence properties hold at the system level. The composition is valid because:

- Intra-process properties are preserved by Appendix I
- Inter-process communication respects capacity bounds (Lemma J.1) and ordering (Lemma J.2)
- The progress invariants and weak fairness ensure the matching construction can always advance (i.e., executions do not diverge via permanent stalling on enabled actions).

Therefore, $\tau\_B,system \approx \tau\_post,system$. □

Remark 1. When all B(c)=0, Sys_B coincides with the rendezvous interpretation of Sys, so the theorem specializes to the rendezvous case.

Remark 2 (Back-annotation and abstract channel boundaries).

The Matchlib library provides a capacity back-annotation feature that extracts finite capacities B(c) from the RTL realization of each message-passing channel and makes them available to the source-level model. In this document, Sys_B is the source-level system consisting of the user-written processes of Sys, interpreted under E4 as abstract bounded-FIFO channels of capacity B(c). (When B(c)=0 this coincides with rendezvous.) The formal results of Appendices J–K are stated for this abstract Sys_B and depend only on the capacities B(c) and the E4 channel semantics at the chosen abstract channel boundaries.

For a simple pipelined loop with a single message-passing input and output, the RTL pipeline's internal staging capacity is accounted for entirely by the back-annotated effective capacity B(c) at the chosen Sys_B channel boundary. Concretely, Sys_B still has exactly one Σ-visible Push_c(v) at the producer endpoint and one Σ-visible Pop_c(v) at the consumer endpoint. With overlapped iterations, the post-HLS RTL may accept/read-ahead up to B(c) values into internal pipeline staging before older iterations produce their corresponding outputs; these accept/read-ahead transfers are modeled as internal microstate (ε) within the concrete realization contributing to B(c). The RTL may realize this staging on either side of a particular module-local "Pop interface" handshake point; that handshake point need not coincide with the chosen Σ-visible Pop endpoint. By construction, the chosen Sys_B boundary encloses all staging counted in B(c) between the Σ-visible endpoints, so movements within that enclosed

realization (including stage-to-stage handoff) do not create additional Σ-visible committed Push/Pop events beyond the single boundary Push/Pop events.

In more complex cases such as HW pipelines with multiple message-passing inputs, the back-annotation mechanism may elaborate the Sys_B (source-level) simulation by introducing additional internal realization structure. See Appendix Q for discussion.

Remark 3 (Practical rendezvous liveness screening under determinism).
When B1 holds and the design's control flow does not branch on empty/full observations of non-blocking interfaces (i.e., it depends only on the ordered history of observable Push/Pop/synchronization actions, not on "probe" outcomes (NB-CF, Appendix I.2)), the rendezvous specialization B(c)=0 is often an effective practical screen for design-level deadlocks: it exercises the most constrained communication discipline and can expose true cyclic data-dependency deadlocks early. Because rendezvous is strictly more constrained than buffered operation, it can also yield false positives: a deadlock observed under B(c)=0 may disappear once finite buffering is present. However, bounded buffering can still introduce capacity-induced deadlocks (FIFO-depth artifacts) when FIFO depths are underestimated; these deadlocks are real for that modeled capacity configuration but may disappear once capacities are increased to the intended design point. Increasing buffer capacity (or applying dynamic buffer-growth strategies) is a standard way to distinguish depth-limited deadlocks from true cyclic dependency deadlocks. Accordingly, the formal results of Appendices J–K deliberately target Sys_B with the actual back-annotated capacities B(c), rather than relying on rendezvous alone.
Remark 4 (When Sys may be used instead of Sys_B for safety-only checks).
If verification is concerned only with safety properties over the chosen DUT-boundary projection Σ (denote it Σ_DUT) (and not with deadlock/liveness), Sys_B remains the default reference model because Σ_DUT includes committed channel-transfer events, and bounded buffering (B(c) > 0) generally changes the set/timing of those committed Push_c/Pop_c events relative to the rendezvous (B(c)=0) case.
Practical note (early bring-up): Before accurate capacities B(c) are available (or before back-annotation can be used), running the rendezvous specialization Sys (B(c)=0) against the intended testbench is still useful to validate functional intent and catch gross protocol/ordering bugs early; however, it remains a screening check, not a proof about the buffered RTL.
Important (soundness): When any channel that can influence Σ_DUT-observable behavior has B(c) > 0, the rendezvous model Sys is often a restriction of Sys_B / RTL_B at Σ_DUT (it can admit fewer Σ_DUT behaviors). Therefore, using Sys in place of Sys_B is generally suitable only as a debug/screening check: a failure can be useful diagnostically, but a pass does not by itself prove Σ_DUT-safety of the buffered implementation.
Sys may be used as a proof-equivalent substitute for Sys_B only when Sys and Sys_B are Σ_DUT-equivalent for the chosen Σ_DUT (i.e., their Σ_DUT-projected trace sets coincide under the intended stimulus). A simple sufficient condition is that every Σ_DUT-visible channel has effective capacity zero at that boundary and that buffering elsewhere is Σ_DUT-opaque (it cannot enable/disable/reorder Σ_DUT-events or change when Σ_DUT-events occur). In all other cases—i.e., if any Σ_DUT-visible channel has B(c) > 0, or if buffered internal channels can affect when Σ_DUT-events occur—use Sys_B with capacities matching RTL_B (Remark 2).
Formal soundness condition. Let Traces_Σ_DUT(M) denote the set of Σ_DUT-projected traces of model M under the intended fixed stimulus. Sys may replace Sys_B for proving a universal Σ_DUT-safety property ϕ only if Traces_Σ_DUT(Sys) = Traces_Σ_DUT(Sys_B); under that condition, Sys ⊨ ϕ iff Sys_B ⊨ ϕ (and otherwise Sys is only a diagnostic/screening check as stated above).

Corollary J.1 (Execution-Level / Trace-Set Form of Theorem J.1)

Purpose This corollary makes explicit the quantifiers implicit in Theorem J.1: Appendix J is intended to relate sets of executions (under a fixed stimulus), not merely to relate a single pre-chosen pre-HLS trace to a single pre-chosen post-HLS trace.

Statement

Fix an initial state and a fixed external stimulus/environment behavior (e.g., the same testbench-driven inputs) for both Sys_B and RTL_B, where Sys_B is the source-level system consisting of the user-written processes of Sys, interpreted under the channel semantics of E4 as abstract bounded FIFOs of capacity B(c) matching RTL_B (see Remark 2 for the abstraction boundary; any back-annotation elaboration is an internal implementation detail). (When B(c)=0 this coincides with rendezvous.)

Clarification (reactive environments). Interpret "fixed stimulus" per Appendix I.2, "Fixed-stimulus interpretation (reactive environments)."

Let Exec_post denote the set of finite execution prefixes of RTL_B that are reachable under this stimulus and that are prefixes of at least one (infinite) execution under the same stimulus that satisfies the Appendix J fairness/progress premises; and let Exec_B denote the corresponding set of finite execution prefixes of Sys_B (defined analogously).

Scope note (intentional): Appendix J is a correspondence result about fair/progress-satisfying infinite behaviors. We adopt the time-divergence / idle-stutter convention B6: from any reachable ε-quiescent state, the system can always be extended to an infinite execution under the same stimulus by appending idle clock ticks that produce no Σ-events. We also note the following stutter-closure/vacuity fact: the Appendix J fairness/progress premises (weak fairness B2 and channel progress invariants B3) constrain only intervals in which some relevant Σ-action/transfer is continuously enabled. Once the ε-quiescent fixed point of B5 is reached (no further internal ε micro-steps are possible and no observable action is enabled), appending idle-stutter ticks does not create any newly enabled observable actions or transfers; therefore B2/B3 remain satisfied (vacuously) on the idle suffix. Consequently, if an ε-quiescent cutpoint is a B5 fixed point, the idle-stutter extension is itself fair/progress-satisfying and the prefix lies in Exec_post / Exec_B. For ε-quiescent cutpoints that are not B5 fixed points (i.e., some observable Σ-action/transfer is enabled), membership in Exec_post / Exec_B depends on the existence of some infinite extension that satisfies B2/B3; idle-stutter alone need not suffice. The qualifier exists to exclude only prefixes that are inconsistent with the stated fairness/progress premises (i.e., prefixes that arise only under unfair scheduler/environment behavior).

Under the assumptions of Theorem J.1 (per-process equivalence from Appendix I, including the witness-compatibility premise WC for ε-modeled non-blocking polls (and any other ε-only internal choices that can affect the next Σ-visible control flow/frontier) as stated in Appendix I.2 and enforceable via Appendix L's snooping / witness-selection technique; NB-CF is a sufficient condition that makes the witness trivial), finite capacities B(c), channel progress invariants P_push / P_pop (B3), and weak fairness (B2)), plus B1/B4/B5 when ε-normalization is needed, the following prefix-matching property holds:

- (Post → B existence) For every $\tau\_post \in Exec\_post$, there exists $\tau\_B \in Exec\_B$ such that $\tau\_B,system \approx \tau\_post,system$ under E1–E5.
- (B → Post existence, RTL-consistent prefixes only) For every $\tau\_B \in Exec\_B$ whose Σ-projection matches a prefix of the RTL_B Σ-trace under the same fixed stimulus (in particular, any τ_B produced by applying Appendix L's snooping wrapper when B1 holds for RTL_B), there exists $\tau\_post \in Exec\_post$ such that $\tau\_B,system \approx \tau\_post,system$.

Moreover, when B1 holds (deterministic **Σ**-projected trace under fixed stimulus), the matching **Σ**-label sequence is unique (up to insertion/removal of finite ε-steps permitted by B4/B5). The corresponding execution prefix witness τ_B need not be unique as a stateful prefix, since Sys_B may admit multiple ε-different realizations with the same Σ-projection. Sys_B may also admit executions whose Σ-projection

does not match RTL_B's Σ-trace when latency-sensitive / non-blocking effects are Σ-visible; such executions are outside the scope of (B → Post) unless constrained by the snooping/witness-selection technique (Appendix L).

Proof (explicit invariant; makes the "projection → witness" bridge explicit)

We prove (Post → B). The (B → Post) clause is identical except that we restrict to RTL-consistent Sys_B prefixes as stated in the corollary. Fix τ_post ∈ Exec_post. Let its Σ-projection be the finite sequence of observable system events:

$e_1\, e_2 \ldots e_n$.

For each i = 0..n, let σ_post[i] be the ε-quiescent RTL_B state obtained by:

(1) taking the prefix of τ_post that contains exactly the first i observable Σ-events, and then

(2) applying maximal finite ε-normalization (B4/B5) to reach a settled (ε-quiescent) cutpoint.

We construct, by induction on i, a Sys_B prefix τ_B[i] ending in Sys_B state σ_B[i], such that τ_B[n] is the desired witness τ_B. The construction maintains the following explicit invariant tying the projection π to the witness. By convention, each σ_B[i] is taken to be an ε-quiescent (ε-normalized) cutpoint as well: after constructing τ_B[i], we (optionally) extend it by a maximal finite ε*-normalization suffix (B4/B5) to reach a settled cutpoint; this does not change τ_B[i],system and therefore does not affect (I1).

Invariant Inv(i).

(I1) Observable-prefix matching:

τ_B[i],system ≈ $(e_1\, e_2 \ldots e_i)$ (equivalently: τ_B[i],system ≈ τ_post,system[0..i)) under E1–E5.

(I2) Projection-to-witness state agreement (per-process, on all Σ-relevant source variables):

For every process P, for every source-level state variable v of P that can influence any future Σ-visible behavior of P (in particular: any v that may be read by BoundaryReq(·,·), by any source-level control predicate Pred_x(·) that decides reachability/selection of the next observable action(s), or by any payload/value expression of such next action(s)),

we have:

v(σ_B[i]) = v(π_P(σ_post[i])),

where π_P(·) is the "standard source-level projection/abstraction" from RTL_B microstate to P's source-level state used in Appendix I. In particular, π_P(σ_post[i]) is an ε-quiescent cutpoint state of P of the kind governed by Lemma I.DR (Deterministic Replay) under fixed stimulus and WC; and σ_B[i] is taken at an ε-quiescent cutpoint as well (by the convention above).

Base case (i = 0).

Let τ_B[0] be the empty Sys_B prefix under the fixed stimulus and σ_B[0] its initial state. By the fixed-stimulus comparison setup, π_P(σ_post[0]) agrees with P's initial source-level valuation. Hence (I1) and (I2) hold.

Inductive step (i → i+1).

Assume Inv(i). Consider the next observable system event e_{i+1} of τ_post,system.

Existence of a matching Sys_B extension.

Let e_{i+1} be the (i+1)-st system-level Σ-event of τ_post,system, and assume Inv(i) holds at ε-quiescent cutpoint σ_B[i] with ε-quiescent σ_post[i]. We show there exists a Sys_B extension

σ_B[i] —ε*→ σ̃_B —e′{i+1}→ σ̂_B —ε*→ σ_B[i+1]

whose Σ-label matches e{i+1} and whose ε* segments are finite, by a case split on the kind of Σ-event e_{i+1}.

> Case 1 (process-local Σ-event of some process P: Push_c(v), Pop_c(v), or an anchored Read/Write event as in E2). Because σ_post[i] is ε-quiescent and e_{i+1} is next on τ_post,system, the corresponding source-level operation of P is enabled at π_P(σ_post[i]) in the sense used by Appendix I's finite-prefix extension argument. By Inv(i).(I2), σ_B[i] agrees with π_P(σ_post[i]) on every source-level variable read by BoundaryReq(·,·), by any control predicate Pred_x deciding reachability/selection of the next observable action(s), and by any

payload/value expression of that next action. Therefore the same request/predicate evaluation holds at σ_B[i], so the corresponding Sys_B operation is enabled as well. Moreover, if the Σ-event carries a value label (Push payload / Pop return / signal value), that label is determined either by the same source-level expression evaluation (Push, Write), by the channel semantics from the matched history (Pop), or by the fixed stimulus plus anchoring discipline (Read), so we may choose the enabled Sys_B operation whose Σ-label equals the Σ-label of e_{i+1}. Finally, by the same Appendix I.4 "finite-prefix extension" reasoning (lifted here to the system cutpoint) together with the corollary's progress premises and weak fairness, Sys_B cannot diverge forever by taking only ε-steps while this matching Σ-action remains continuously enabled; hence there exists a finite ε* segment reaching σ̃_B where the matching Σ-action commits, producing e′_{i+1}.

Case 2 (rendezvous-style matched pair, if B(c)=0 or equivalently synchronized Push/Pop is modeled as a single Σ-event). Let P and Q be the participating endpoint processes. Enablement of the rendezvous on τ_post,system at σ_post[i] implies that both endpoints' boundary requests/predicates are satisfied at π_P(σ_post[i]) and π_Q(σ_post[i]). By Inv(i).(I2), the same endpoint conditions hold at σ_B[i], so the rendezvous is enabled in Sys_B as well. By weak fairness/progress (and Appendix I.4's rendezvous case), Sys_B cannot take an infinite ε-only divergence while the rendezvous remains continuously enabled; therefore there exists a finite ε* after which the rendezvous Σ-event commits with the same Σ-label as e_{i+1}.

Case 3 (explicit synchronization-call Σ-event, or other explicit sync). Enablement transfers from π_P(σ_post[i]) to σ_B[i] by Inv(i).(I2) applied to the relevant source control state. Finite completion follows by the same "enabled ⇒ eventually taken" argument under weak fairness and bounded stutter.

In all cases, there exists a finite ε* Sys_B extension realizing a Σ-event e′{i+1} whose label matches e{i+1}. Let τ_B[i+1] be τ_B[i] extended by this ε* ∘ e′_{i+1} segment, and then (optionally) further extended by a maximal finite ε*-normalization suffix to reach the ε-quiescent cutpoint σ_B[i+1] (B4/B5). Then (I1) holds for i+1 by construction.

Preservation of the projection-to-witness state agreement (I2) via deterministic replay (Lemma I.DR). Fix any process P. Let $(e_1\ e_2\ \ldots\ e_{i+1})|P$ denote the P-local Σ-label sequence induced by the matched system prefix $e_1\ e_2\ \ldots\ e\{i+1\}$ (i.e., the projection of that system prefix onto P's Σ-events, preserving the Σ-labels under E1–E5). By (I1) and the definition of ≈ (per-process closure), the same P-local Σ-label sequence $(e_1\ e_2\ \ldots\ e_{i+1})|P$ is realized by the Sys_B witness prefix τ_B[i+1] as well.

Under the fixed stimulus and the WC premise, apply Lemma I.DR (Deterministic Replay) to: (a) P's execution within the chosen Sys_B witness prefix τ_B[i+1] reaching the ε-quiescent cutpoint σ_B[i+1], and (b) the source-level replay corresponding to the RTL_B prefix reaching the projected ε-quiescent cutpoint π_P(σ_post[i+1]). These two executions start from the same initial source-level state and have the same P-local Σ-label sequence $(e_1\ e_2\ \ldots\ e\{i+1\})|\_P$. Therefore Lemma I.DR implies the resulting ε-quiescent source-level cutpoint states agree; in particular, for every Σ-relevant source-level variable v of P we have:

v(σ_B[i+1]) = v(π_P(σ_post[i+1])).

Because P was arbitrary, (I2) holds at i+1. Thus Inv(i+1) is established.

Conclusion.

By induction, Inv(n) holds. Taking τ_B = τ_B[n] yields the required witness prefix for (Post → B), and clause (I2) is the explicit invariant that ties the RTL_B projection π to the constructed Sys_B witness state. This is the state-level bridge that Corollary J.2 relies on when it interprets σ_post via projection and then derives frontier/guard/value correspondence at ε-quiescent cutpoints. □

Corollary J.2 (State Correspondence at the End of a Matched Observable Prefix)

Purpose This corollary serves as the bridge between trace equivalence and state correspondence needed in Appendix K. The mapping target is the source-level bounded-FIFO semantics of the design (capacity B(c)), not an unrelated rendezvous (B(c)=0) execution state.

Statement

Assume the hypotheses of Corollary J.1 (equivalently: the system-level assumptions of Theorem J.1 for Sys_B vs RTL_B under the fixed stimulus), and additionally assume B1 holds for RTL_B (unique Σ-projected trace under the fixed stimulus), plus B4 and B5 for ε-normalization. The Σ-label sequence is therefore unique (up to insertion/removal of finite ε-steps), although the Sys_B witness prefix/state need not be unique as a full micro-state.

Let τ_post be any τ_post ∈ Exec_post (as defined in Corollary J.1).

Define its **ε**-normalization $\bar{\tau}$_post as τ_post extended by a (possibly empty) finite suffix ε* to a terminal state $\bar{\sigma}$_post that is ε-quiescent, meaning: no further ε-step is enabled from $\bar{\sigma}$_post (so any further progress, once enabled, must proceed via an observable Σ-action rather than additional internal ε-steps). Take ε* to be a maximal finite ε-extension of τ_post; such a maximal finite extension exists under B4 (finite stutter bound). Moreover, "canonical endpoint" is meant modulo the source-level projection used below: by B5 (quiescence closure), any two maximal finite ε-extensions of the same observable prefix reach ε-quiescent endpoints that agree under this projection (and therefore yield the same source-level variables / NextFront / logical FIFO state for purposes of clauses (1)–(2)). Hence $\bar{\sigma}$_post is well-defined (up to ≡) as the ε-quiescent representative for the observable prefix.

(In clauses (1)–(2) directly below, interpret σ_post as $\bar{\sigma}$_post, i.e., the ε-normalized terminal state. When referring below to "source-level variables," the "next observable frontier," and the "logical FIFO state" in σ_post, interpret those notions via the standard source-level projection/abstraction from RTL_B microstate to the corresponding source-level state as used in Appendix I; micro-architectural registers and bookkeeping are ignored by this projection.)

Let Sys_B denote the source-level system consisting of the user-written processes of Sys, interpreted under the channel semantics of E4 as abstract bounded FIFOs of capacity B(c) matching RTL_B (see Remark 2 for the abstraction boundary; any back-annotation elaboration is an internal implementation detail). (When B(c)=0 this coincides with rendezvous.) Then there exists a finite execution prefix τ_B of Sys_B ending in some reachable state σ_B such that:

(1) The observable prefixes match: τ_B,system ≈ τ_post,system (equivalence under E1–E5).

(2) The terminal states correspond at the source level: σ_B ≡ σ_post, where "≡" means:

(Note: "≡" is intentionally a frontier/enable/value correspondence (clauses (a)(b)(c)), not full micro-state equality; internal pipeline/buffering microstate may differ between σ_B and σ_post provided such differences are ε-steps that are WFG-inert as assumed in Appendix K.)

(a) For every process P, NextFront_P(σ_B) = NextFront_P(σ_post), where NextFront_P(σ) is as defined in Lemma S2 (the set of minimal (w.r.t. P's source partial order ≤_P) observable items that may occur next). Concretely, this means either:

- a (possibly multi-element) set of pending message actions (Push_c and/or Pop_c) on distinct interfaces that are currently minimal candidates and may be issued and (if channel-enabled) committed in any order or together in one cycle, or
- an explicit synchronization call s, together with the same set of signal Read/Write actions that are logically anchored at s per E2.
- This frontier equality does not require the pre-HLS and post-HLS models to commit identical subsets of that message set in the same cycle; it only identifies the set of currently-minimal candidates.

Note. NextFront_P is a candidate frontier relative to the fixed witness universe Σ_P of Lemma S2 and does not by itself assert enablement or "pending blocking" status. Appendix K reasons

about pending blocking transfers using Front_P(σ); see Lemma K.0 (K.2.2) for the Front/NextFront bridge at ε-quiescent cutpoints.

(b) By Lemma S1, for every process P, and for every item x ∈ NextFront_P(σ_B) (equivalently x ∈ NextFront_P(σ_post) by clause (a)):

(i) The evaluation of x's boundary request (and any source-level control predicate used to decide that x is the next observable action) is the same in σ_B and σ_post; equivalently, σ_B and σ_post agree on all source-level state variables read when evaluating that guard/predicate. (Channel-side enablement of Push/Pop is handled separately by clause (c).)

(ii) If x produces a value—i.e., x is a Push_c, or x is a signal Write action (or set of such actions) anchored at the next synchronization call—then the value produced by x (payload/value expression) is the same in σ_B and σ_post (hence σ_B and σ_post agree on all source-level variables read by that expression). If x observes a value via signal Read actions anchored at the next synchronization call, those observed signal values are the same in σ_B and σ_post because the stimulus is the same and E2 anchors the reads to that same synchronization call.

(iii) If x is a Pop_c, then the value returned by that Pop_c is determined by the channel's logical FIFO contents after the matched observable prefix; by E4 and the matched Push/Pop history, that logical head element is the same for σ_B and σ_post.

(c) For every channel c, the abstract FIFO state in σ_B (empty/full predicate, and the logical head element when non-empty) agrees with the logical FIFO state induced by the matched Push_c/Pop_c history of (1). Physical micro-architectural realization—pipeline registers, arbitration bookkeeping, and the implementation of buffering—is abstracted away by "≡", but the logical occupancy/ordering facts induced by the matched trace are not. Equivalently, letting occ_c(t) denote the logical Σ-boundary occupancy used throughout Appendix K (and letting "empty/full" refer to this same logical occupancy), σ_B ≡ σ_post implies σ_B and σ_post agree on occ_c(t) and on the logical head element when non-empty.

In particular, internal transfers within the concrete realization (e.g., movement between pipeline/buffering stages) that do not commit a boundary transfer are treated as ε and do not change this abstract FIFO state. The logical head/occupancy at the chosen Σ boundary advances only on Σ-visible boundary commits (Push_c/Pop_c) that appear in the matched Push/Pop history; equivalently, occ_c(t) is a function only of that matched boundary-commit history (and B(c)), and bounded-FIFO legality is checked solely at this boundary using the annotated B(c).

Lemma S1 (Relevant source-level state agreement for next-frontier guards/payloads)

Purpose.

This lemma is the minimal per-process "state agreement" fact needed to justify Corollary J.2, clause (2b): matched observable prefixes determine (enough of) the source-level state so that the next observable frontier's request/guard evaluation and produced values agree, allowing Appendix K to reason about BoundaryReq/blockedness on the Sys_B side.

Statement.

Fix a process P. Let τ_post,P be the projection of some reachable post-HLS (RTL_B) finite prefix onto P, and let τ_B,P be a Sys_B prefix such that τ_B,P ≈ τ_post,P under the document's Σ-matching/E1–E5 conventions. Let σ_post be the ε-normalized (ε-quiescent) terminal state of τ_post,P, and let σ_B be the terminal state of τ_B,P.

Terminology (Witness-compatible; WC).

Assume this comparison τ_B,P ≈ τ_post,P is witness-compatible in the sense of WC (Appendix I): any ε-modeled non-blocking poll outcome (and any other ε-modeled internal choice that can affect P's next Σ-

visible control flow / next Σ-visible frontier) is fixed/selected consistently with the compared RTL_B prefix τ_post,P (e.g., via Appendix L's witness-selection/snooping wrapper).
Remark (NB-CF as a sufficient condition). If the design satisfies NB-CF (Appendix I)—failed PopNB/PushNB polls do not influence the next Σ-visible control flow/frontier—then the witness is trivial and WC holds automatically.
Let NextFront_P(σ) denote P's "next observable source-level frontier" (the minimal candidate set w.r.t. P's source partial order, as used in Corollary J.2). Let x be any item in that next frontier (so x ∈ NextFront_P(σ_B), and equivalently x ∈ NextFront_P(σ_post) by the frontier-alignment lemma (S2) / Corollary J.2(2a)).
Then:
(1) Guard / boundary-request agreement (local enablement side).
BoundaryReq(x, σ_B) = BoundaryReq(x, σ_post).
Moreover, for any source-level control predicate Pred_x that is evaluated (in the source semantics) to decide that x is the next observable action at this point (e.g., branch/loop condition determining control reachability of x), we have:
Pred_x(σ_B) = Pred_x(σ_post).
Equivalently: σ_B and σ_post agree on the valuation of every source-level state variable read when evaluating BoundaryReq(x,·) and Pred_x(·).
(2) Produced-value agreement (payload/value expressions).
If x produces a value (i.e., x is a Push_c(v), or x is a signal Write(sig,val) (or set of such writes) anchored at the next synchronization call), then the value produced by x is the same in σ_B and σ_post.
Equivalently: σ_B and σ_post agree on the valuation of every source-level state variable read by x's payload/value expression.
(3) Pop return-value agreement.
If x is a Pop_c, then the value returned by that Pop_c is the same in σ_B and σ_post, and is equal to the logical head element of c's abstract FIFO after the matched observable prefix (hence uniquely determined by the matched Push/Pop history under the bounded-FIFO semantics).
(4) Signal-read agreement at the next sync anchor (when applicable).
If x observes values via signal Read(sig,·) actions anchored at the next synchronization call, then those observed signal values are the same in σ_B and σ_post under the "fixed stimulus" interpretation used by Appendix I/J together with the E2 anchoring discipline.
Proof.
We prove (1)–(4) by reducing them to a single per-process invariant: "matched Σ-prefixes fix the valuation of the source-level variables relevant to the next frontier," plus FIFO legality for Pop values.
Define the sequence of P-local observable events in the matched prefixes:
τ_post,P = e1 e2 … en
τ_B,P = e1 e2 … en
(where equality here is equality of Σ-labels under the document's event matching convention: same channel/signal identity, and same value label for Push/Pop/Read/Write events).
For k = 0..n, let:
• σ_post[k] be the ε-quiescent post-HLS state after the first k P-local observable events.
• σ_B[k] be the Sys_B state after the first k P-local observable events.
(Existence of these ε-quiescent representatives is exactly the ε-normalization discipline used throughout Appendix I/J.)
Key invariant I(k).
For every source-level state variable v of process P that can influence any future Σ-visible behavior of P (in particular: any v that may be read by (i) BoundaryReq(·), (ii) a control predicate Pred_x deciding reachability/selection of the next frontier item(s), or (iii) a payload/value expression for the next frontier

item(s)), we have:
v(σ_B[k]) = v(σ_post[k]).
(Here σ_post[k] is interpreted via the "standard source-level projection/abstraction" from RTL_B microstate to source-level state used in Appendix I. In particular, this projected source-level cutpoint state is the one used for deterministic replay in Lemma I.DR: under fixed stimulus and the WC premise, it is uniquely determined by the preceding P-local Σ-label sequence.)

Deterministic replay (Lemma I.DR).
Under the fixed stimulus and the WC premise, Lemma I.DR gives source-state determinacy at ε-quiescent cutpoints: for a fixed witness, the source-level state after the first k P-local Σ-events is uniquely determined by the initial source-level state together with the P-local Σ-label prefix (including any Pop return value or anchored Read value).
Apply Lemma I.DR to the two executions induced by the compared prefixes: (a) the Sys_B execution of P producing τ_B,P, and (b) the source-level replay corresponding to τ_post,P under the fixed witness (WC), whose ε-quiescent cutpoints are exactly the projected states σ_post[k]. These two executions start from the same initial source-level state (fixed-stimulus setup) and produce the same P-local Σ-label sequence e1 e2 … en (by Σ-label matching). Therefore, for every k, the source-level cutpoint states agree; in particular, every source-level state variable v covered by I(k) satisfies v(σ_B[k]) = v(σ_post[k]). Thus I(k) holds for all k, and in particular I(n) holds.

Conclusion of the invariant.
By induction, I(n) holds at the terminal matched cutpoints σ_B = σ_B[n] and σ_post = σ_post[n].
Now discharge the lemma's four conclusions for any x ∈ NextFront_P(σ_B):
(1) BoundaryReq(x,·) and Pred_x(·) are source-level predicates evaluated from source-level variables of P at the cutpoint. Since those variables agree by I(n), the evaluations agree:
BoundaryReq(x, σ_B) = BoundaryReq(x, σ_post) and Pred_x(σ_B) = Pred_x(σ_post).
(2) Any payload/value expression for a Push or anchored Write reads some set of source-level variables of P. Those variables agree by I(n), so the produced value agrees.
(3) For Pop_c, the returned value is the FIFO head after the matched history; this is identical on both sides by the bounded-FIFO legality semantics and the matched Push/Pop history.
(4) For anchored signal reads at the next synchronization call, the observed values agree under the fixed-stimulus interpretation plus anchoring; this is the same mechanism used to justify matching Read(sig,val) labels in the first place.
Therefore, (1)–(4) hold, proving Lemma S1. ■

Lemma S2 (Prefix-to-frontier determinacy / "matched Σ-prefix ⇒ same next observable frontier")

Fix a process P. Let τ_B,system be a finite Sys_B execution prefix ending in state σ_B, and let τ_post,system be a finite RTL_B execution prefix ending in state σ_post. Assume:
(A1) τ_B,system ≈ τ_post,system under E1–E5.
(A2) σ_post is the ε-normalized (ε-quiescent) endpoint associated with τ_post,system (as used in Corollary J.2).
(A3) ≤_P is P's source-induced partial order on Σ_P generated by the document's R-rules (R1–R4) and the chosen channel/synchronization semantics (i.e., the same ≤_P is used on both Sys_B and RTL_B sides).
Definition (Next observable candidate frontier).
For any terminal state σ, define NextFront_P(σ) as follows.

1. Let τ_P(σ) be the projection of the system prefix leading to σ onto P's Σ-events. Clarification (dynamic Σ-event universe; fixed for this comparison). In this lemma, Σ_P denotes the set of P's dynamic Σ-event occurrences in the per-process Σ-projection determined by the fixed external

stimulus for the comparison τ_B,system ≈ τ_post,system (including, when applicable, any witness-selected non-blocking outcomes required by Appendix I.2). Thus Σ_P is a single shared universe for both prefixes (Sys_B and RTL_B): it does not depend on how a finite prefix is extended, nor on ε-interleavings between Σ-events. In particular, Σ_P excludes Σ-events on untaken control-flow paths (e.g., the "else" branch when the "if" branch was taken) and excludes Σ-events from loop iterations that are not executed in the fixed stimulus- (and witness-) determined per-process Σ-projection used for this comparison. Note: even if one of the compared prefixes is deadlocked and therefore cannot realize further committed Σ-events, Σ_P remains this fixed witness universe and is not reduced by deadlock.

2. Let Done_P(σ) ⊆ Σ_P be the set of Σ-events of P that occur in τ_P(σ), using the document's canonical per-endpoint/per-signal/per-sync occurrence matching (e.g., "k-th Push_c matches k-th Push_c", "k-th Sync matches k-th Sync", etc.). (Well-definedness for channel events: under ≈, this canonical per-endpoint occurrence matching preserves ordinal indices, so "k-th Push_c/Pop_c" refers to the same matched Σ-event on both sides.)
3. Let Remain_P(σ) := Σ_P \ Done_P(σ).
4. Define: NextFront_P(σ) := min_{≤_P}(Remain_P(σ)), i.e., the set of ≤_P-minimal remaining Σ-events for P. This is exactly Corollary J.2's "set of minimal (w.r.t. P's source partial order) remaining observable items" and should be read as a *candidate* next observable frontier relative to the fixed witness universe Σ_P. In particular, because Σ_P is not reduced by deadlock (note above), membership in NextFront_P(σ) does not assert that the event is enabled or realizable as an immediate successor of σ when σ is deadlocked; it only asserts ≤_P-minimality among the not-yet-seen items in Σ_P.

Claim.
Under (A1)–(A3),
NextFront_P(σ_B) = NextFront_P(σ_post).
Proof.
Step 0 (Reduce to the per-process view).
From τ_B,system ≈ τ_post,system (A1), and because ≈ (E1–E5) is defined componentwise over the per-process projections (see Appendix I's definition of ≈ and its per-process closure), the per-process projections satisfy:
τ_B,P ≈ τ_post,P.
It therefore suffices to show that Done_P(σ_B) = Done_P(σ_post); frontier equality then follows immediately from the shared Σ_P (by the definition above) and the shared ≤_P (A3).
Step 1 (Synchronization events completed are identical).
Under the document's canonical matching, synchronization events for P are matched by source order / occurrence index (the k-th sync in the source corresponds to the k-th sync in the observable history). Under E1 (and the sync-order constraints embedded in ≈), τ_B,P and τ_post,P contain the same set (equivalently, the same prefix) of P's sync events. Hence the "current sync interval" is the same on both sides.
Step 2 (Anchored signal I/O events completed are identical).
Signal reads/writes are matched canonically by (signal, occurrence index), with their anchoring to sync points enforced by E2. Since Step 1 shows the same sync indices have been traversed, and E2 preserves the anchoring structure, τ_B,P and τ_post,P contain the same set of P's Σ-visible signal read/write events. Thus signal events contribute the same elements to Done_P(σ_B) and Done_P(σ_post).
Step 3 (Committed message events completed are identical).
By the Σ-event matching convention built into the definition of ≈, committed channel events are paired canonically by (channel endpoint, ordinal occurrence): for each channel c, the k-th committed Push_c in τ_B matches the k-th committed Push_c in τ_post (and likewise for Pop_c), with identical payload/value

labels. The commutations/regroupings permitted by ≈ only affect the interleaving and same-cycle grouping of Σ-events on distinct independent endpoints; they do not change the per-endpoint projected sequence of committed events (i.e., for any fixed endpoint e, proj_e(τ_B,P) and proj_e(τ_post,P) are identical as sequences of Σ-labels up to the common prefix length). Therefore, for every endpoint-indexed committed event (e,k) of P, that event occurs in τ_B,P iff it occurs in τ_post,P, with the same label. Hence τ_B,P and τ_post,P contain exactly the same set of P's committed Σ-visible channel events, and message events contribute the same elements to Done_P(σ_B) and Done_P(σ_post).
Step 4 (Conclude Done equality, then frontier equality).
From Steps 1–3:
Done_P(σ_B) = Done_P(σ_post).
Therefore:
Remain_P(σ_B) = Σ_P \ Done_P(σ_B) = Σ_P \ Done_P(σ_post) = Remain_P(σ_post).
Since ≤_P is the same relation on both sides (A3), taking *≤P-minimal elements preserves equality: min{≤P}(Remain_P(σ_B)) = min{≤_P}(Remain_P(σ_post))*.
By definition of NextFront_P(·), this is:
NextFront_P(σ_B) = NextFront_P(σ_post).
QED.

Proof of Corollary J.2

Step 0 (By definition, work with **ε**-normalized terminal states) By the ε-normalization defined in the Statement, w.l.o.g. assume τ_post ends in an ε-quiescent state (no ε-step is enabled); we write this terminal state as σ_post. (If not, replace τ_post by any maximal finite ε-extension $\bar{\tau}$_post = τ_post ∘ ε*, which preserves the observable prefix and is finite by B4. By B5, the choice of maximal ε* does not matter for the reasoning below: all such ε-quiescent endpoints agree under the source-level projection (hence are interchangeable for ≡ / NextFront / FIFO-state conclusions), so we treat the endpoint as the canonical ε-quiescent representative.)
Step 1 (Get a matching source-level observable prefix) Fix the external test stimulus that produced the given post-HLS prefix τ_post. By the system-level prefix-matching property (Corollary J.1) applied to Sys_B and RTL_B, there exists a finite execution prefix τ_B of Sys_B under that same stimulus such that τ_B,system ≈ τ_post,system under E1–E5. Let σ_B be the terminal state of this witness prefix τ_B.
Step 2 (Align per-process control points) By Lemma S2, because τ_B,system ≈ τ_post,system and E1/E2/E3/E4/E5 preserve per-process same-interface order, no-reverse constraints, side-of-sync constraints, and signal anchoring at synchronization points, each process P has executed the same per-endpoint observable history in both prefixes (up to commutation/regrouping of independent distinct-interface message actions). Therefore, at the end of the prefixes, NextFront_P(σ_B) = NextFront_P(σ_post) (clause (2a)), including the same logically anchored signal Read/Write actions at the next synchronization call when applicable.
Step 3 (Align the relevant source-level state) By Lemma S1, matched prefixes preserve the source-level state needed for next-action enablement and payloads exactly as in Appendix I, giving clause (2b). Clause (2c) holds because, under E4, the abstract FIFO empty/full status and logical head element are uniquely determined by the matched Push/Pop history of (1), and thus coincide in σ_B and σ_post.
Combining Steps 1–3 gives a terminal source-level state **σ**_B with **σ**_B ≡ **σ**_post. □

J.3 Causal Dependency ("Happens-Before") and What System-Level Equivalence Guarantees (Informative)

The system-level result of Appendix J establishes equivalence of the *aggregate* observable behavior, i.e. $\tau\_B,system \approx \tau\_post,system$ over the alphabet Σ of channel operations, synchronization events, and signal Read/Write actions.

This equivalence is intentionally *observational*: it constrains what an external environment can distinguish at Σ, not internal RTL micro-timing.

As a consequence, Appendix J should be read as preserving functionally significant ordering, not incidental latency. Functionally significant ordering is exactly the ordering induced by the happens-before relation ($\rightarrow$) defined in the "Causal Dependency Between Processes" section: a designer must express any required cross-process ordering using message-passing edges, explicit synchronization (e.g., barriers), or explicit signal handshakes; designs "shall not rely" on the relative order of causally independent events.

Accordingly, the system-level theorem implies the following *causality preservation* principle:

- If two synchronization anchors s1 and s2 are causally related ($s1 \rightarrow s2$), then the post-HLS execution preserves their order. This is because each elementary happens-before link is preserved: intra-process order is preserved by the per-process result; message-passing links are preserved by Lemma J.2 ($clk\_post(Push\_c) \leq clk\_post(Pop\_c)$); and explicit synchronization/handshake links propagate only at clock edges under the same signal and synchronization semantics. By transitivity, the entire causal chain preserves order.
- If s1 and s2 are causally independent (neither $s1 \rightarrow s2$ nor $s2 \rightarrow s1$), then their relative order is *not* constrained by the methodology, and differences are treated as incidental latency variation; relying on such ordering is outside the formal guarantees.

This perspective is what makes the "single testbench / no surprises" implication precise: any testbench that observes only Σ and encodes its expectations through causal dependencies (channels, barriers, or explicit handshakes) cannot distinguish Sys_B (the pre-HLS model interpreted with bounded-FIFO capacities B(c) matching RTL_B) from the post-HLS RTL. (The rendezvous specialization Sys with B(c)=0 is covered only under the explicit conditions of Remark 4.)

Appendix K – Liveness Preservation for Buffered Implementations

K.1 Scope, Notation, and Definition of Internal Message-Passing Deadlock

Terminology alignment. Throughout Appendix K, we use the Common Formal Definitions predicates: 'boundary request is asserted' means BoundaryReq(m,t); 'channel-enabled' means ChannelEnabled(m,t); and 'channel-disabled' means ChannelDisabled(m,t) (all evaluated at the ε-quiescent ready/valid fixed point for cycle t).

We work with a fixed design:

- System Sys_B: The collection of pre-HLS processes obeying the R-rules and B-rules, interpreted under the source-level channel semantics of E4 as abstract bounded FIFOs. Each channel c has capacity $B(c) \geq 0$ (the same B(c) as in the RTL implementation, see Remark 2 in Appendix J).
- Post-HLS implementation RTL_B: The hardware implementation of the same processes in which each channel c is realized with finite capacity $B(c) \geq 0$. In this appendix, we compare Sys_B (source-level bounded-FIFO semantics) to RTL_B (micro-architectural realization). The proof does not require mapping buffered RTL states to a distinct rendezvous (B(c)=0) state.

- Witness-selection note (debug/simulation): Appendix K's liveness argument uses Corollary J.2 only to assert existence of a corresponding Sys_B state σ_B for a given ε-normalized RTL state σ_post (via the source-level correspondence ≡). If, for practical debug, one wishes to construct a deterministic Sys_B execution that tracks the unique RTL Σ-trace (B1) so that σ_B/WFG_B are reproducible, then the Appendix L snooping wrapper may be applied to select a witness Sys_B execution aligned to RTL latency-sensitive events.

A global state σ of either system consists of:

1. For each process P: a control location (program point) and local state.
2. For each channel c: its abstract bounded-FIFO state at the Sys_B channel boundary—its current occupancy occ_c(t) and (for buffered channels) its ordered contents. Here occ_c(t) counts all items that have been committed by Push_c but have not yet been committed by Pop_c at that abstract boundary. In RTL_B, this count includes items residing anywhere in the concrete realization of channel c that contributes to the back-annotated capacity B(c) (e.g., FIFO storage, skid/elastic buffers, and any HLS-inserted pipeline staging that can hold in-flight messages). Thus Sys_B treats that entire contributing realization as a single composite buffer at one Sys_B boundary; internal stage-to-stage movements inside that realization (including FIFO→pipeline-stage handoffs) are ε-steps and do not create additional committed Push_c/Pop_c Σ-events beyond the boundary events counted by occ_c.
3. Any additional architectural state not representing buffered messages on any channel (e.g., computation pipeline registers, arbitration state, bookkeeping, etc.).

We adopt the usual Wait-For Graph (WFG) abstraction, restricted to internal message-passing channels. Internal means: the producer and consumer of channel *c* are both processes of *Sys_B* / *RTL_B* (i.e., both endpoints are in the modeled system).
Channels whose complementary endpoint is the external environment/testbench (DUT boundary) are excluded from the WFG and from the deadlock definition in Appendix K.

- Vertices: Processes.
- Edges: There is a directed edge P→Q via channel c in state σ if there exists some operation op ∈ Front_P(σ) on channel c such that op is channel-disabled (i.e., ChannelDisabled(op,t) holds), and Q is the unique process that is the complementary endpoint for op (producer/consumer partner for c). This edges definition applies uniformly for both buffered channels (B(c)>0, enable is space/data availability via occ_c) and rendezvous channels (B(c)=0, enable requires the complementary endpoint's boundary request to be true in the same cycle), hence WFGs may contain a mix of edges via both kinds of channels. Here Front_P(σ) is the set of minimal (w.r.t. P's program-order / partial source order) pending blocking channel operations op∈{Push_c,Pop_c} (issued by P but not yet committed); Front_P(σ) may contain multiple operations on distinct interfaces (reflecting the R/E freedom to issue/commit multiple Push/Pop in one cycle). P is "blocked on message passing" in σ as defined in Definition (Blocked on message passing) (K.1). Note: WFG edges may exist even when P is not blocked (e.g., one frontier op is channel-disabled while another is channel-enabled); however, internal message-passing deadlock (K.1) considers only sets D in which every P∈D is blocked, so all WFG edges relevant to deadlock reasoning originate from blocked processes with no channel-enabled frontier operation. When Front_P(σ) has multiple members, P may have multiple outgoing WFG edges (one per blocked frontier op).
  (No deassert-before-commit assumption.) Blocking Push_c/Pop_c requests are non-withdrawable: once the request corresponding to a frontier operation is asserted, it is not deasserted before that operation commits (and Push_c payload remains stable while asserted). This rules out "try-and-withdraw" behavior for blocking transfers within the WFG abstraction.
  By Assumption A6 (point-to-point channels), the complementary endpoint Q for any

blocked Push_c/Pop_c on c is unique.

Shared-resource note: If the RTL contains sharing that would otherwise make the complementary endpoint non-unique (e.g., multi-producer or multi-consumer access to a logical resource), that sharing is modeled as an explicit arbiter process connected to each client by point-to-point channels, so WFG edges remain well-defined.

All WFG reasoning is over **ε**-quiescent (**ε**-normalized) global states: internal pipeline/arbitration micro-steps are treated as ε and are not represented as separate WFG blocking points. Here, "ε-quiescent" includes the fixed point of purely combinational ready/valid propagation, so K.1 enablement is evaluated only on those settled ready/valid values.

(WFG-inert **ε** premise.) We assume these ε-steps are observationally silent with respect to the WFG abstraction: for any $\sigma \xrightarrow{\varepsilon^*} \sigma'$, they do not change (i) the potentially-blocking frontier Front_P(σ) for any process P (i.e., which guarded blocking Push/Pop operations are minimal and pending), nor (ii) the truth of any boundary requests relevant to those frontier operations, nor (iii) the channel-side enabling status (as defined in K.1) of any op ∈ Front_P(σ). Equivalently, ε-steps do not change the channel state relevant to K.1 enabling (e.g., occ_c(t) for B(c)>0 buffered channels), and they do not create/remove rendezvous enablement for a frontier transfer except via Σ-visible committed Push_c/Pop_c events.

In particular (pipelined loops / overlapped iterations): when HLS introduces overlap that may initiate later-iteration message reads "early," we assume the design obeys the Automatic flush (drain-only blocking discipline) defined in Appendix B. Under that discipline, overlapped execution does not introduce any later-iteration blocking Push/Pop into Front_P(σ), so transient internal pipeline activity remains ε and does not contribute wait-for edges. Overlapped execution in pipelined loops is modeled only as a potential retiming of ordinary Σ-visible boundary commits (Push_c/Pop_c) relative to the unpipelined source, and is accounted for by the back-annotated bounded-FIFO semantics at the chosen Sys_B boundary (E4). In particular, the channel facts relevant to K.1 enabling/WFG edges—e.g., occ_c(t) for B(c)>0, and rendezvous enablement for B(c)=0—change only on Σ-visible committed Push_c/Pop_c events at that boundary; internal pipeline movement/staging within the composite realization counted in B(c) is ε and does not affect K.1 enablement or introduce/remove WFG edges.

Precondition (Kε: WFG-inert ε / no hidden micro-timing control decisions). All results in Appendix K are stated under the precondition Kε that the WFG-inert ε premise above holds for the design. If an implementation intentionally uses internal micro-timing to change such control decisions, then Kε is violated unless that behavior is made Σ-observable (e.g., via explicit modeling/snooping) or is witness-selected under the Appendix I.2 framework.

A common instance is branching on the success/failure of a non-blocking poll in a way that changes the next Σ-visible candidate frontier NextFront_P(**σ**) (Corollary J.2 / Lemma S2); in that case, either NB-CF must hold (Appendix I.2), or the poll outcomes must be witness-selected as stated there.

Non-blocking polling attempts and other purely internal actions do not contribute edges to the WFG; they are modeled as internal ε-steps. When a non-blocking call succeeds, that success is already represented at the Σ level as the corresponding committed Push_c/Pop_c event, but it does not itself add a wait-for edge because it is not a blocking operation.

SyncChannels and barrier well-formedness.

SyncChannel (barrier) operations are treated as pure synchronization events and are omitted from the Wait-For Graph, which tracks only blocking Push/Pop dependencies. We therefore interpret "deadlock" in this appendix as internal message-passing deadlock (i.e., a deadlocked set in which all processes are blocked on some internal Push/Pop dependency captured by the WFG). Any global stall at a barrier

caused by mismatched or conditional participation (e.g., a process can permanently bypass the barrier or reach it a different number of times) is a specification-level error already present in the pre-HLS model; such stalls are not created by the HLS transformation and are excluded by an explicit barrier well-formedness assumption (A8 below), not by the internal message-passing deadlock premise. Under this assumption, omitting SyncChannels from the WFG is sound: a post-HLS deadlock implies the existence of a Push/Pop wait-cycle as characterized below.

Definition (Blocked on message passing). Let Front_P(σ) denote the set of minimal (w.r.t. P's program-order / partial source order) pending blocking channel operations op ∈ {Push_c, Pop_c} (i.e., operations issued by P but not yet committed). We say that P is blocked on message passing in state σ iff Front_P(σ) is non-empty and, for every op ∈ Front_P(σ), BoundaryReq(op, σ) holds and ChannelDisabled(op, σ) holds (equivalently, BoundaryReq(op, σ) ∧ ¬ChannelEnabled(op, σ)). Note (ALL disabled ⇒ stall). If any frontier op is channel-enabled at the chosen Σ-visible boundary, then P is not "blocked on message passing" for WFG purposes. Consequently, an implementation may not be physically stalled at the chosen Σ boundary by a cross-channel "join" predicate while some frontier Σ-boundary op remains channel-enabled; any such conjunctive/join dependency must be represented as an explicit staged/bundle/join boundary channel so that, when the process is physically stalled, the relevant frontier ops on that explicit boundary are all channel-disabled.

An internal message-passing deadlock is a reachable state σ in which there exists a non-empty set of processes D such that:

- Every process P in D is blocked on message passing, and in particular there exists at least one op ∈ Front_P(σ) on an internal message-passing channel c (i.e., a channel included in the WFG) such that BoundaryReq(op, σ) holds and ChannelDisabled(op, σ) holds.
- The vertices in D, together with the internal channels they access, form a strongly connected component in the WFG that has no outgoing edges (a closed cycle): processes in D can only wait on each other.

Intuitively, D is a set of processes that are waiting only on each other (via internal message-passing channels tracked by the WFG) and can never be unblocked by activity elsewhere in the modeled system. A process that is blocked solely on channels whose complementary endpoint is the external environment/testbench (excluded from the WFG) does not witness an internal message-passing deadlock in this appendix.
Our notion of liveness in this appendix is: Liveness = absence of internal message-passing deadlock as defined above. (Starvation of a single process in an otherwise live system is ruled out separately by weak fairness.)

K.2 Assumptions and Imported Results

K.2.1 Standing Assumptions We assume throughout Appendix K:
A1. R-rules and B-rules. The source-level model Sys_B and the post-HLS implementation RTL_B satisfy the process- and channel-level semantic rules defined earlier (R-rules and B-rules).
A2. Finite Pipeline Depth (FPD). There is a global bound on the number of purely internal steps (ε-steps) that can occur between consecutive observable actions.
A3. Weak Fairness (WF) (B2). If an action remains continuously enabled (its guard remains true), the scheduler eventually selects it.
A4. Channel Progress (B3; Appendix N). For each channel c with capacity B(c), assume the progress invariants of Appendix N hold. This is an obligation on the scheduler/environment (e.g., fairness of arbiters/back-pressure) and is not implied by the local R-rules alone:

• P_push(c): If a process P is stalled at a blocking Push_c with a persistent (non-withdrawable; no deassert-before-commit) request asserted, and the complementary endpoint process Q continuously has the matching Pop_c pending as a frontier operation (i.e., Pop_c ∈ Front_Q(σ) continuously, with its boundary request (i.e., BoundaryReq(Pop_c,t)) remaining true), then the transfer on c must eventually complete (in particular: for B(c)>0, some complementary Pop_c eventually commits, freeing space; for B(c)=0, the rendezvous completion eventually occurs).
• P_pop(c): If a process P is stalled at a blocking Pop_c with a persistent (non-withdrawable; no deassert-before-commit) request asserted, and the complementary endpoint process Q continuously has the matching Push_c pending as a frontier operation (i.e., Push_c ∈ Front_Q(σ) continuously, with its boundary request (i.e., BoundaryReq(Push_c,t)) remaining true), then the transfer on c must eventually complete (in particular: for B(c)>0, some complementary Push_c eventually commits, providing data; for B(c)=0, the rendezvous completion eventually occurs).
A5. Source-Level Liveness Assumption. The source-level bounded-FIFO system Sys_B (with capacities B(c) as defined in K.1) is deadlock-free under assumptions A1–A4.
A6. Point-to-point channels (single-producer/single-consumer). For each message-passing channel c (including buffered channels with B(c)>0 and rendezvous channels with B(c)=0), exactly one process performs all Push_c operations on c (the producer) and exactly one process performs all Pop_c operations on c (the consumer). Hence the complementary endpoint for any blocked Push_c/Pop_c is unique, and WFG edges are well-defined even for mixed WFGs that include both buffered and rendezvous channels.
If a design has a shared resource that would violate this uniqueness, it must be modeled via an explicit arbiter process plus point-to-point channels (as noted in K.1).
A7 (Determinism / witness selection). Assume B1 holds for RTL_B (post-HLS), so the Σ-labeled execution under the given stimulus is unique up to finite ε-stutter.
Do not require B1 for Sys_B: there may be multiple Sys_B executions that are ≡ to the same τ_post at the Σ/frontier level. Appendix K uses only the existence of such a Sys_B witness (Corollary J.2).
When a deterministic, reproducible Sys_B witness is desired for simulation/debug, apply the Appendix L snooping wrapper to select a particular witness aligned with the latency-sensitive (arbiter-visible) ε-choices. We also rely on B4 and B5 (Appendix N) to justify **ε**-normalization / **ε**-quiescent endpoints, as required by Corollary J.2 (state correspondence).
A8. Barrier well-formedness (SyncChannels). For each SyncChannel instance, the set of designated participant processes reaches the barrier the same number of times under the fixed stimulus/initial state; equivalently, no participant can permanently bypass a barrier call while another participant is waiting at that barrier. (Barrier stalls are therefore outside the behaviors considered in Appendix K's internal message-passing deadlock analysis.)
A9. Single-transfer-per-cycle endpoint constraint (scalar channel interface). For every message-passing channel c, in any clock cycle t, at most one Push_c commit and at most one Pop_c commit may occur on c. If a design uses multi-lane/burst transfers on a logical resource, it must be modeled explicitly (e.g., k parallel point-to-point channels or a widened message) so that the Σ-level per-channel event model satisfies this constraint.
A10. WFG-inert **ε** (K**ε**). The design satisfies Precondition Kε (WFG-inert ε / no hidden micro-timing control decisions) as stated in K.1.

K.2.2 Lemma K.0 — Front/NextFront Bridge at **ε**-Quiescent Cutpoints

Purpose. Appendix K defines WFG edges using Front_P(σ) (minimal pending blocking Push/Pop operations), while Corollary J.2's state correspondence σ_B ≡ σ_post is phrased in terms of NextFront_P(σ) plus BoundaryReq agreement for items in that next frontier. This lemma relates these two frontier notions at the ε-quiescent cutpoints where Appendix K reasons.

Statement. Fix an ε-quiescent global state σ (of either Sys_B or RTL_B) and a process P. Define the message-action slice of the next observable frontier:
MsgNextFront_P(σ) ≜ { x ∈ NextFront_P(σ) | x is a message action Push_c or Pop_c }.
Under the Issue/Commit linkage (well-formedness) and the no-deassert-before-commit discipline of K.1 for blocking transfers, the pending blocking frontier equals the asserted blocking slice of that next message frontier:
Front_P(σ) = { x ∈ MsgNextFront_P(σ) | BoundaryReq(x,σ) }.
Consequently, for any corresponding ε-quiescent states (σ_B,σ_post) with σ_B ≡ σ_post (Corollary J.2), we have Front_P(σ_B)=Front_P(σ_post), and for every op ∈ Front_P(σ_B) the boundary request BoundaryReq(op,·) agrees in σ_B and σ_post.
Proof. Fix an ε-quiescent global state σ and a process P.
(⊆) Let op ∈ Front_P(σ). By definition of Front_P(σ) (Appendix K.1), op is a pending blocking message operation (issued by P but not yet committed) that is minimal with respect to P's program-order / partial source order among pending blocking message operations.
First, BoundaryReq(op,σ) holds. This follows from Issue/Commit linkage and the no-deassert-before-commit discipline for blocking transfers: once a blocking Push/Pop is issued, its endpoint-owned request remains asserted (with stable payload for Push) until that operation commits, and the assertion remains attributed to that same outstanding operation at ε-quiescent cutpoints until an intervening commit occurs on that endpoint direction.
Second, op ∈ MsgNextFront_P(σ). Since op has not yet committed, the corresponding Σ-visible message action has not occurred in τ_P(σ), so op ∈ Remain_P(σ).
Moreover, by blocking-call control-flow semantics together with the drain-only/automatic-flush discipline (Appendix B, "Drain-only downstream progress," formalized in the ≤_P / issue sense), a pending blocking op cannot be bypassed: while op remains pending, P does not issue any new blocking message operation instance y such that op <_P y. Therefore, no Σ-visible message action y with y <_P op can remain uncompleted at σ; otherwise y's corresponding operation instance would still be pending at σ, contradicting the ≤_P-minimality of op within Front_P(σ). Hence op is among the ≤_P-minimal remaining Σ-visible message actions, i.e., op ∈ MsgNextFront_P(σ).
Thus op ∈ { x ∈ MsgNextFront_P(σ) | BoundaryReq(x,σ) }.
(⊇) Let x ∈ MsgNextFront_P(σ) and assume BoundaryReq(x,σ)=true. By Issue/Commit linkage (stable attribution while outstanding), an asserted endpoint-owned request at an ε-quiescent cutpoint is attributed to a unique outstanding issued message operation on that endpoint direction, and that attribution cannot change without an intervening commit on that endpoint direction. Since BoundaryReq(x,σ)=true and x is a message action in the next observable candidate frontier, x is exactly that outstanding issued operation instance and has not yet committed in σ; hence x is pending. Since x is a ≤_P-minimal remaining Σ-visible message action (x ∈ MsgNextFront_P(σ)), it is, in particular, a minimal pending blocking message operation among those currently outstanding in the interval. Therefore x ∈ Front_P(σ).
Combining both directions yields:
Front_P(σ) = { x ∈ MsgNextFront_P(σ) | BoundaryReq(x,σ) }.
Finally, for corresponding ε-quiescent states (σ_B,σ_post) with σ_B ≡ σ_post (Corollary J.2), we have NextFront_P(σ_B)=NextFront_P(σ_post) and BoundaryReq agreement for all x ∈ NextFront_P. Applying the characterization above on each side yields Front_P(σ_B)=Front_P(σ_post), and BoundaryReq agrees for every op in that common Front set. □

K.3 Lemma K.1 — Characterization of Internal Message-Passing Deadlock (Buffered and Rendezvous Cases)

Statement: Let σ_post be a reachable ε-quiescent global state of the post-HLS system RTL_B (equivalently, the ε-normalization of some reachable state, which exists as a finite extension by B4/B5). Suppose there is a non-empty set of processes D that is internally message-passing deadlocked (as defined in K.1). Then:

Every process P in D is blocked on a blocking channel operation (Push or Pop), not on an internal step.

Buffered-channel case (B(c) > 0):

• If P is blocked on Push_c and B(c) > 0, then occ_c(t) = B(c) (channel is full).

• If P is blocked on Pop_c and B(c) > 0, then occ_c(t) = 0 (channel is empty).

Rendezvous case (B(c) = 0):

• If P is blocked on Push_c and B(c) = 0, then the complementary endpoint is not simultaneously requesting the matching Pop_c in σ_post (i.e., its Pop_c boundary request is not true in that cycle, so rendezvous enabling for this transfer is false).

• If P is blocked on Pop_c and B(c) = 0, then the complementary endpoint is not simultaneously requesting the matching Push_c in σ_post.

The processes in D form a closed strongly connected component in the WFG.

Proof:

Internal Steps: Because σ_post is ε-quiescent (i.e., an ε-normalized endpoint that exists as a finite extension by B4/B5), no process has any enabled internal ε-step in σ_post. In particular, if some process had an internal step whose guard is true in σ_post, then σ_post would not be ε-quiescent: the ε-normalization could be extended by at least one additional ε-step. Therefore, no process in D is blocked on an internal step with a true guard; since each P ∈ D is blocked on message passing (K.1), the blocking operation must be a blocking channel operation (Push or Pop).

Blocked Push: By the definition of "blocked on message passing" in K.1, if P is blocked due to a frontier Push_c (i.e., Push_c ∈ Front_P(σ_post)), then Push_c's boundary request is true and the channel-side enabling condition for Push_c is false.

• If B(c) > 0, the channel-side enabling condition for Push_c is space availability, i.e., occ_c(t) < B(c). Hence, if P is blocked on Push_c in σ_post, necessarily occ_c(t)=B(c) (channel is full).

• If B(c) = 0 (rendezvous), the channel-side enabling condition is "the complementary Pop_c endpoint's boundary request is true in the same cycle" (K.1). Since P is blocked, this rendezvous enabling condition is false; equivalently, the complementary endpoint is not simultaneously requesting Pop_c in σ_post.

Blocked Pop: Symmetric. By the definition of "blocked on message passing," if P is blocked due to a frontier Pop_c (i.e., Pop_c ∈ Front_P(σ_post)), then Pop_c's boundary request is true and the channel-side enabling condition for Pop_c is false.

• If B(c) > 0, the channel-side enabling condition for Pop_c is data availability, i.e., occ_c(t) > 0. Hence occ_c(t)=0 (channel is empty).

• If B(c) = 0 (rendezvous), the channel-side enabling condition is "the complementary Push_c endpoint's boundary request is true in the same cycle" (K.1). Since P is blocked, this rendezvous enabling condition is false; equivalently, the complementary endpoint is not simultaneously requesting Push_c in σ_post.

Closed Cycle: By definition of internal message-passing deadlock, processes in D cannot wait on processes outside D, or else an external action could unblock them.

K.4 Lemma K.2 — Dependency Refinement

We now relate the blocking dependencies in the RTL buffered system to those in the source-level bounded-FIFO semantics Sys_B.

Statement: For any pair of corresponding states (σB,σpost) where σB≡σpost (Corollary J.2), the wait-for edges agree in both directions: for any processes P,Q, there is a wait-for edge P→Q in WFGpost iff there is a wait-for edge P→Q in WFGB. (In particular, WFGpost⊆WFGB, and the reverse inclusion holds as well.) This holds for mixed WFGs containing edges via both buffered channels (B(c)>0) and rendezvous channels (B(c)=0).
Proof: Since σB≡σpost, clause (2a) gives NextFront alignment for every process P, and clause (2b) gives BoundaryReq agreement for every x ∈ NextFront_P at this ε-quiescent cutpoint. By Lemma K.0 (Front/NextFront Bridge), it follows that the pending blocking frontiers coincide: for every P, Front_P(σB)=Front_P(σpost), and BoundaryReq(op,σB)=BoundaryReq(op,σpost) for every op ∈ Front_P. Moreover, for buffered channels with B(c)>0, the channel empty/full predicate agrees between σB and σpost (Corollary J.2, clause (2c)). For rendezvous channels with B(c)=0, the rendezvous enabling predicate ("the complementary endpoint's boundary request is true in the same cycle") also agrees. The only subtlety is that σB≡σpost gives BoundaryReq agreement directly only for items in NextFront; to transfer the complementary endpoint's rendezvous-request predicate, we use the Issue/Commit linkage plus Lemma K.0: at an ε-quiescent cutpoint, if the complementary endpoint's BoundaryReq for the rendezvous partner is true, then that request is attributed to a unique outstanding issued blocking partner operation which (by the same blocking/persistence discipline used in Lemma K.0) is ≤-minimal among remaining message actions, hence lies in Front of that complementary process and is therefore covered by the established Front alignment / BoundaryReq agreement.
We show that each RTL edge transfers to the source-level WFG by case-splitting on B(c).
Case 1: Buffered channel (B(c)>0).
Consumer (Blocked Pop): Suppose P waits for Q via Pop_c in the RTL system on a channel c with B(c)>0. By Lemma K.1, occ_c(t)=0 in σpost (empty). By σB≡σpost and Corollary J.2(2c) (which equates the abstract FIFO state with the logical Σ-boundary occupancy occ_c(t)), we have the same logical occupancy in σB; hence occ_c(t)=0 in σB as well. Under the bounded-FIFO channel semantics, completion of P's Pop_c requires a complementary Push_c by Q. Hence P→Q is also an edge in WFGB.
Producer (Blocked Push): Symmetric. If P waits for Q via Push_c on a channel c with B(c)>0, Lemma K.1 gives occ_c(t)=B(c) (full) in σpost. By σB≡σpost and Corollary J.2(2c), σB has the same logical Σ-boundary occupancy; hence occ_c(t)=B(c) in σB as well. Completion of P's Push_c requires a complementary Pop_c by Q to make space. Hence P→Q is also an edge in WFGB.
Case 2: Rendezvous channel (B(c)=0).
Producer (Blocked Push): Suppose P waits for Q via Push_c in the RTL system on a channel c with B(c)=0. By Lemma K.1, the complementary endpoint Q is not simultaneously requesting Pop_c in σpost, i.e. BoundaryReq(Pop_c,σpost)=false in that cycle. Suppose for contradiction that BoundaryReq(Pop_c,σB)=true. By Issue/Commit linkage (stable attribution of asserted requests at ε-quiescent cutpoints), this implies there exists a unique outstanding issued Pop_c instance op_Q for Q with BoundaryReq(op_Q,σB)=true. By the same blocking/persistence discipline used in Lemma K.0 (no bypass / no deassert-before-commit for blocking transfers), an asserted outstanding blocking message operation must be ≤_Q-minimal among remaining message actions; hence op_Q ∈ Front_Q(σB). By Front alignment under σB≡σpost (Lemma K.0's consequence) and BoundaryReq agreement for frontier items, we obtain BoundaryReq(op_Q,σpost)=true, contradicting BoundaryReq(Pop_c,σpost)=false from Lemma K.1. Therefore BoundaryReq(Pop_c,σB)=false. Hence Push_c is channel-disabled in σB, and P→Q is an edge in WFGB.
Consumer (Blocked Pop): Symmetric. Suppose P waits for Q via Pop_c on a channel c with B(c)=0. By Lemma K.1, the complementary endpoint Q is not simultaneously requesting Push_c in σpost, i.e. BoundaryReq(Push_c,σpost)=false in that cycle. Suppose for contradiction that BoundaryReq(Push_c,σB)=true. By Issue/Commit linkage, this implies there exists a unique outstanding issued Push_c instance op_Q for Q with BoundaryReq(op_Q,σB)=true. By the same blocking/persistence

discipline used in Lemma K.0, op_Q must be ≤_Q-minimal among remaining message actions; hence op_Q ∈ Front_Q(σB). By Front alignment under σB≡σpost and BoundaryReq agreement for frontier items, BoundaryReq(op_Q,σpost)=true, contradicting Lemma K.1. Therefore BoundaryReq(Push_c,σB)=false. Hence Pop_c is channel-disabled in σB, and P→Q is an edge in WFGB. Therefore every RTL wait-for dependency is also a source-level wait-for dependency, so WFGpost⊆WFGB.
Conversely, suppose P→Q is an edge in WFGB. By the edge definition in K.1, there exists some operation op ∈ Front_P(σB) on channel c such that op is channel-disabled in σB and Q is the unique complementary endpoint for c. From the Front/BoundaryReq alignment established above (via σB≡σpost and Lemma K.0), we have op ∈ Front_P(σpost); and from the buffered (empty/full via occ_c) and rendezvous (complementary BoundaryReq) enabling-predicate agreement established above, op is also channel-disabled in σpost. Hence P→Q is an edge in WFGpost. Therefore WFGpost and WFGB have exactly the same wait-for edges (in both directions) at corresponding cutpoints. □

K.5 Theorem K.1 — Liveness Preservation

Statement: If the source-level bounded-FIFO system Sys_B is deadlock-free and the Appendix I/J correspondence prerequisites hold (in particular NB-CF, or else RTL-consistent witness-selected NB outcomes for latency-sensitive NB control flow), then RTL_B is also deadlock-free.
This theorem preserves absence of message-passing deadlock (WFG over blocking Push/Pop). SyncChannel/barrier deadlocks are treated as specification errors and are out of scope here.
Proof (by Contradiction):

1. Assume RTL_B reaches a deadlocked state **σ**_post.
2. Exec_post applicability at a deadlock cutpoint: By the time-divergence / idle-stutter convention (B6), the finite reachability prefix ending at σ_post can be extended (under the same stimulus) to an infinite execution. It remains to justify that there exists an extension that satisfies the Appendix J fairness/progress premises (B2/B3). At σ_post, we are at an ε-quiescent cutpoint and (by the definition of internal message-passing deadlock in K.1) there exists a non-empty set D of processes such that every P ∈ D is blocked on message passing and D is closed in the WFG (processes in D can only wait on each other). In particular, for every P ∈ D and every op ∈ Front_P(σ_post), BoundaryReq(op,σ_post) holds and ChannelDisabled(op,σ_post) holds. Since blocking-call control-flow semantics forbid bypassing a pending blocking frontier op, no Σ-visible message transfer belonging to any P ∈ D is enabled at σ_post; and by closure of D, actions of processes outside D cannot make any such frontier op of a process in D become channel-enabled. Hence weak fairness (B2), which constrains only actions that are continuously enabled, imposes no obligation to schedule any frontier-blocking transfer of a process in D, since those transfers are never enabled on any continuation that preserves this deadlock. For processes outside D, choose an infinite continuation under the same stimulus that satisfies the fairness/progress premises in the usual way (i.e., whenever some observable Σ-action/transfer remains continuously enabled, it is eventually taken; and if the run reaches a B5 fixed point, the remainder may be idle-stutter). Moreover, the channel-progress invariants (B3) apply only when their antecedents hold; for channels incident to D, Lemma K.1's full/empty / rendezvous-request characterization ensures the complementary endpoint request needed to trigger the B3 antecedent is not simultaneously present at σ_post for any frontier-blocking transfer of a process in D. Therefore there exists an infinite fair/progress-satisfying extension of the prefix reaching σ_post, so this prefix lies in Exec_post, and Corollary J.2 applies at this ε-quiescent cutpoint.

3. Extract Cycle: Fix a non-empty set of processes D as guaranteed by the definition of internal message-passing deadlock in K.1: (i) every process P ∈ D is blocked on message passing at σ_post, and (ii) D forms a strongly connected component in WFG_post with no outgoing edges (i.e., D is closed: processes in D can only wait on each other). In the remainder of this proof, "deadlocked process" means "a process in D". We invoke Lemma K.1 only after fixing this D, to characterize what each P ∈ D is blocked on (and the associated full/empty / rendezvous-request facts) at σ_post, including when checking whether any B3 antecedent could hold for a frontier-blocking transfer of a process in D.
4. Map to Source-Level Buffered Semantics: By Corollary J.2 (State Correspondence at the End of a Matched Observable Prefix), there exists a corresponding source-level state σ_B of Sys_B such that σ_B ≡ σ_post.
5. Transfer Cycle: By Lemma K.2, for this corresponding pair (σB,σpost) the wait-for edges agree in both directions: for any processes P,Q, P→Q is an edge in WFG_post iff P→Q is an edge in WFG_B. Since D is a closed SCC in WFG_post, every outgoing wait-for edge from any P∈D targets a process in D. Therefore every outgoing wait-for edge from any P∈D in WFG_B also targets a process in D, so D has no outgoing edges in WFG_B and forms a closed wait-for cycle there as well.
6. Contradiction: This implies Sys_B is deadlocked in state σ_B, contradicting the Source-Level Liveness Assumption (A5).
7. Conclusion: RTL_B cannot deadlock. □

Appendix L – Snooping Introduction, Implementation and Concerns

Snooping Introduction

If designs are completely latency insensitive, verification of the pre-HLS versus post-HLS models is straightforward. However, if the design contains some components such as arbiters which have latency-sensitive behavior, and if that latency-sensitive behavior is in some cases externally visible to the DUT, then verification becomes somewhat more complex. Techniques can be applied to simplify verification.

Consider a design which has an arbiter like Matchlib toolkit example 09*. The arbiter uses non-blocking PopNB() on its inputs, and blocking Push() on its output to emit the winner of the arbitration. Since the latency between the pre-HLS and post-HLS designs will differ, the order of transactions presented to the arbiter in the two scenarios will differ, and thus the order of the winners will differ. This may result in verification mismatches between the two models if the order of the winners is externally visible to the DUT.

To force the pre-HLS and post-HLS simulations to match, we can run the two designs side-by-side. We can snoop the inputs to the arbiter in the post-HLS model, and only allow the inputs to the pre-HLS arbiter to proceed when the corresponding inputs are seen in the post-HLS model. This will force the order of the inputs to be equivalent between the pre-HLS and post-HLS models, and thus both arbiters will pick the same winners. When this technique is used, the pre-HLS simulation will be throttled by the post-HLS simulation, but the overall verification will still work properly.

Another related example involves interrupt request signals feeding into an interrupt controller within a CPU model. Typically, each interrupt request signal is a single bit sc_signal<>, indicating that an interrupt request is pending. If the requests originate from accelerator blocks that are being synthesized through HLS, then because of the latency differences between the pre-HLS and post-HLS models, the order of interrupt requests arriving at the interrupt controller will differ between the two models, and this will likely result in verification mismatches if the differences are externally visible to the DUT. To force the order of the requests to match, we snoop the requests arriving at the controller in the post-HLS model, and only then allow the requests to be seen in the pre-HLS model.

Simplified Snooping Approaches

In the snooping example directly above in which the arbiter is snooped, the entire post-HLS RTL DUT is simulated alongside the pre-HLS model to force the arbiter requests to be aligned in time. This is the most general case and assumes the input delays to the arbiter are difficult to determine via analysis.

Sometimes there are simpler cases where the input delays to the arbiter in the RTL DUT are easier to determine. For example, each input to the arbiter might arrive a fixed number of clock cycles after one of the primary inputs to the RTL DUT is pushed by the testbench. In this case, a much simpler model can be used to align the pre-HLS model with the post-HLS model. We can simply monitor the primary inputs to the pre-HLS model and then apply the fixed cycle delays to the pre-HLS arbiter inputs.

The two cases above illustrate two ends of a spectrum of possible approaches for extracting the delays from the RTL DUT. Between these two points there exist other possible approaches.

Snooping Implementation

To enable perfect matching between the pre-HLS and post-HLS system simulations, latency-sensitive global signals and latency-sensitive non-blocking message-passing operations need to be synchronized between the two simulations if their latency differences are externally visible to the DUT.

As explained earlier in this appendix, in the general case the entire post-HLS DUT RTL must be run alongside the pre-HLS system to achieve alignment. Examples of signals that need to be synchronized include global interrupt request signals (as discussed earlier in this appendix) and rdy/vld signal pairs for non-blocking operations on message-passing channels. All such synchronization signals and their corresponding handshake signals must be level-stable:

- A latency-sensitive signal s is level-stable if, once s becomes 1 in cycle t, it must remain 1 until the corresponding handshake signal h is 1 in some cycle $t' \geq t$.

Once the set of signals that need to be aligned between the two simulations is identified, the general rule to implement snooping is simple:

- For each of the pre-HLS and post-HLS simulations, make all readers of the signals see the logical AND of the values of each signal being driven in each separate simulation.

Often the post-HLS simulation will never run ahead of the pre-HLS simulation, because HLS typically only adds (rather than subtracts) latency within each process. This means that often the only signals that need to be delayed are on the pre-HLS side, and therefore the logical AND of the nets may not need to

be driven on the post-HLS side. If this optimization technique is used, the logical AND of the nets should be compared with the actual value driven on the post-HLS side, and if they do not perfectly match then the optimization technique must not be used.

Snooping Concerns

The snooping technique is used to align pre-HLS and post-HLS latency-sensitive global signals and latency-sensitive non-blocking message-passing operations in cases where their latency differences are externally visible at the DUT boundary. When such a wrapper is applied, the pre-HLS model is throttled to follow the latency choices made by the post-HLS RTL, and the resulting traces at the observable interfaces are forced to agree.

Readers with a formal background may be concerned by this technique, because it appears to use the post-HLS RTL implementation to modify the behavior of the pre-HLS specification. From a formal point of view, the important observation is that the pre-HLS model is intentionally under-specified with respect to micro-timing/latency: subject to the R-rules, B-rules, weak fairness, and the happens-before relation on Σ (committed IO events; unsuccessful non-blocking polls are ε-steps), it can admit multiple ε-/latency-different executions that respect the same happens-before constraints. This is compatible with B1, which asserts that the post-HLS Σ-projected trace is unique under a fixed stimulus; the pre-HLS side may still have multiple admissible realizations that (when projected to Σ) match that unique RTL Σ-trace.

Snooping does not change what executions are allowed by the pre-HLS specification. Instead, it selects—purely for verification purposes—one admissible pre-HLS execution whose latency-sensitive events align with those observed in the RTL. In other words, the implementation is used to pick a witness execution of the specification, not to redefine the specification.

Experienced SoC architects tend to view this slightly differently, in terms of design tradeoffs and verification cost.

First, it is usually possible at the architectural level to avoid latency-sensitive behaviors entirely, for example by insisting that all communication be latency-insensitive and fully synchronized. However, the quality-of-results (QOR) costs of such designs may be unacceptable. Introducing latency-sensitive behavior—non-blocking arbiters, latency-sensitive global interrupt signals, and so on—is a deliberate choice made by the designer to meet performance, power, or area goals.

Second, in many practical systems, latency-sensitive behavior is not functionally observable at the DUT boundary under the equivalence relation ≈ with Σ instantiated to include only the DUT-boundary observables (Appendix G; Common Formal Definitions above). As an example, consider a DUT that contains a single-port RAM used to exchange data between two internal processes. An arbiter is required to control access to that RAM port, and the arbiter uses latency-sensitive non-blocking Pop operations on its request channels. During HLS, internal latencies will change, so the order in which the arbiter sees pending requests and chooses winners may differ between the pre-HLS and post-HLS models.

In this example:

- The individual RAM read/write operations and the internal arbitration decisions are not directly visible at the DUT interface.
- Only the higher-level IO of the two processes (for example, their message-passing interfaces to the rest of the system) is externally visible.

One might initially conclude that it is necessary to snoop the arbiter's inputs to make the pre-HLS and post-HLS traces match. However, the modeling rules impose two additional constraints:

1. Weak fairness for the arbiter. The arbiter must satisfy the weak fairness assumptions, so that any request that remains pending is eventually granted in both the pre-HLS and post-HLS models.
2. Happens-before discipline on shared memory. The system must enforce a happens-before relation on shared-memory accesses, ensuring that sufficient synchronization exists between the two processes so that there are no read/write races through the RAM in either model.

Under these conditions, the different arbitration orders merely change the relative timing of internal operations and of causally independent external events. They do not change the functional ordering of causally dependent observable actions at the DUT boundary. In the terminology of Appendix G, the differences are incidental variations in latency rather than changes to the happens-before partial order, and therefore they are not considered observable differences in behavior at the DUT boundary. In such cases, snooping is not required. (See also Note 1 below).

(More precisely: the equivalence relation $\approx$ is parameterized by the chosen observable alphabet Σ. If internal functionality (such as the above RAM arbiter) uses channels/signals that are not designated observable in the chosen instantiation of Σ (see Common Formal Definitions), then mismatches on those internal actions are outside $\approx$ by construction: they are not in Σ. If those internal channels/signals are designated observable (e.g., by snooping for debug, or by expanding Σ for Appendix K deadlock analysis), then they are Σ-events and must match under $\approx$. In addition, even when B1 holds (unique post-HLS Σ-trace under fixed stimulus), the pre-HLS model may still admit multiple ε-/latency-different executions consistent with the same happens-before constraints; Corollary J.1 captures the intended quantification ("existence of a matching witness execution prefix") rather than requiring a single pre-chosen trace.)

Experienced SoC architects therefore try to avoid designs in which latency-sensitive internal behaviors leak directly into the observable behavior of the system under test, since this both complicates verification and makes RTL behavior less predictable. When they do introduce such behaviors—examples include non-blocking arbiters whose winners affect externally visible ordering, global interrupt request signals at the DUT boundary—they typically know exactly where those latency-sensitive interfaces are and can isolate them.

A useful analogy is a subsystem whose clock frequency is adjusted dynamically based on on-die temperature. As temperature changes, the externally visible behavior of the SoC can change (for example, in terms of throughput or timing of events), and designers may choose such a temperature-dependent scheme to meet overall system requirements. To verify such a system, we may wish to compare a concrete RTL trace captured at a specific temperature profile with a higher-level reference model. To do so, the reference model must be driven with the same temperature evolution as the RTL saw. If that temperature profile is not otherwise under direct control, the simplest way to obtain it is to "snoop" the temperature as observed in the RTL trace and replay it into the reference model.

In this analogy, temperature is an external parameter of the environment rather than a fundamental part of the functional specification, but it still influences observable behavior. Snooping simply provides a mechanism to recover that parameter from the implementation when it cannot be easily prescribed a priori. Similarly, snooping of latency-sensitive IO provides a mechanism to supply implementation-specific latency choices—subject to the fairness and happens-before constraints—back to the pre-HLS specification so that the two models can be compared under a common environment.

Note 1 (Unobservable non-blocking micro-timing).
Safety / functional equivalence. The equivalence relation ≈ is parameterized by the chosen observable alphabet Σ (Appendix G). If, for safety-only checking, we instantiate Σ to include only DUT-boundary observables (Σ_DUT) and we assume that any latency-sensitive internal behavior is not visible at that boundary under ≈—i.e., it may change only internal micro-timing, but it does not change the values, ordering, or presence of Σ_DUT events—then snooping of those internal latency-sensitive interfaces is not required for proving Σ_DUT-safety. Any mismatches on actions outside Σ_DUT are outside ≈ by construction.
Liveness / deadlock analysis. For message-passing deadlock analysis (Appendix K WFG), any latency-sensitive interface whose ε-level choices can affect whether a blocking Push/Pop is enabled, or can create/remove a wait-for dependency without an intervening committed Σ-event, must be accounted for. Practically, the default is to snoop and replay every such latency-sensitive (typically non-blocking) arbitration/probe interface in the DUT. Snooping can be avoided only if the design is certified "WFG-inert" with respect to that interface: it cannot change WFG edges except via committed Push_c/Pop_c (or explicit synchronization) events.

Stress Testing Pre-HLS Models for Latency Robustness

The formal equivalence results in Appendices G–K establish that, under the scheduling rules and fairness assumptions, the post-HLS RTL is trace-equivalent to the pre-HLS model at all observable interfaces. These results implicitly assume that the pre-HLS specification is *well-formed*: it must not rely on incidental, implementation-specific timing alignments for functional correctness. Designs that do rely on such incidental alignments fall outside the formal guarantees.

Designs that use non-blocking message-passing operations (e.g., PushNB/PopNB, ac_channel nb_read/nb_write) or latency-sensitive global signals are particularly vulnerable to this kind of fragility. For such designs, it is strongly recommended to subject the pre-HLS model to aggressive *latency stress tests* in which message and handshake latencies are varied across executions. The intent is to validate that the design behaves correctly across the range of weakly fair schedules permitted by the modeling rules, not just under one convenient execution order.

More concretely, verification should check that:

- Causal dependencies are explicit. All functionally significant "happens-before" relationships are enforced via message-passing, SyncChannel operations, or explicit handshake protocols, rather than by relying on the incidental execution order of concurrent processes. In the terminology of Appendix G, the design shall not depend on the ordering of causally independent events for correctness.
- Weak fairness does not hide latent deadlocks or starvation. The design continues to make progress under adversarial but *weakly fair* scheduling—i.e., enabled actions may be delayed

arbitrarily long but not forever—consistent with the B2/B3 obligations on the scheduler and environment summarized in Appendix N.

If the pre-HLS model fails under such randomized-latency stress scenarios, then the specification itself is outside the formal model of this document: it violates the design rule that well-formed systems must not rely on the relative ordering of causally independent events. In that situation, the equivalence theorems still apply to well-formed traces, but they no longer guarantee that the "golden" specification is robust under the full range of latency variations allowed by the environment.

The Matchlib library provides practical mechanisms to automate these stress tests in pre-HLS simulation, including:

- Random stall injection on message-passing channels, to simulate variable communication delays while preserving rendezvous semantics.
- Latency and capacity back-annotation, to model specific per-channel buffer depths and delays, including the capacities B(c) chosen during HLS.

Worked examples of this methodology are provided in Catapult Matchlib examples 60* and 72*, which illustrate how to configure randomized stalls and back-annotated latencies when validating that a design remains well-formed and latency-robust.

Appendix M – Document Abstract

This document defines a precise user-level scheduling model for high-level synthesis (HLS) that enables system-level verification and debug to be carried out almost entirely on the pre-HLS SystemC model, while still permitting aggressive RTL optimizations in the synthesized design. It introduces a small set of uniform rules governing three classes of I/O operations—message-passing channels, signal I/O, and explicit synchronization—and organizes them into a "basic conceptual model" that preserves source-order synchronization, pins signal reads and writes to their nearest synchronization points, and constrains the reordering of message-passing operations so that HLS cannot introduce new deadlocks.

These rules are then extended to pipelined loops, shared and external memories (via an explicit array-access mapping layer and conflict-free reordering), and "direct input" pragmas that allow stable or periodically synchronized signals to bypass unnecessary internal storage while maintaining equivalence between pre- and post-HLS behavior.

The methodology is latency-insensitive by construction but accommodates latency-sensitive islands such as cycle-accurate transactors, non-blocking arbiters, and one-way handshake protocols through encapsulation and carefully specified synchronization schemes. The document also provides concrete modeling guidelines (e.g., rules for placing signal reads/writes around wait statements, coding of rolled loops with signal I/O, and use of Matchlib Connections and SyncChannel) that allow digital verification engineers to write a single testbench in SystemVerilog UVM or SystemC/C++ and reuse it across both pre-HLS and post-HLS models with "no surprises."

A major contribution is the formalization, in Appendix G and subsequent appendices, of a trace-equivalence relation between the pre-HLS source model and the post-HLS RTL, expressed as partial-

order constraints on observable I/O actions and encapsulated in equivalence rules E1–E5. For channels whose RTL implementations introduce finite buffering, the correspondence is stated against a source-level bounded-FIFO interpretation Sys_B whose capacities B(c) match RTL_B; the rendezvous case B(c)=0 is recovered as a special case.

These proofs are compositional (per process and system-level), incorporate weak fairness and bounded-FIFO assumptions, and cover key HLS transformations including loop pipelining, added FSM states, memory-access reordering, and configurable channel buffering. Collectively, the scheduling rules, coding guidelines, and formal guarantees provide a tool-agnostic, Catapult-compatible foundation for industrial HLS use and a concrete starting point for standardization efforts within bodies such as the Accellera Synthesis Working Group.

Keywords:
High-level synthesis (HLS); SystemC; scheduling rules; latency-insensitive design; message-passing channels; signal I/O; loop pipelining; direct input pragmas; shared memory; trace equivalence; formal verification; bounded FIFOs; Catapult HLS; Matchlib; Accellera Synthesis Working Group.

Appendix N - Scheduler and Environment Fairness Assumptions

The formal results in Appendices I–K rely on several semantic assumptions about the post-HLS scheduler and its environment:

- B2 (Weak fairness of the scheduler). If the enabling predicate for an action (Push, Pop, Sync) remains continuously true from some cycle onward, the scheduler must eventually select that action within a finite, but unspecified, number of cycles.
- B3 Channel Progress Invariants (ready/valid form).
  For every channel c with capacity B(c), interpret "continuously enabled" in terms of continuous request, not as an immediate commit.
  (No deassert-before-commit assumption.) For blocking Push_c/Pop_c transfers, these requests are non-withdrawable: once asserted for a transfer attempt, they are not deasserted prior to the corresponding commit (and Push_c payload remains stable while vld_c is asserted).
  Let vld_c denote the producer's persistent request to transfer (e.g., valid=1 with a stable payload for a Push_c), and let rdy_c denote the consumer's persistent request to transfer (e.g., ready=1 for a Pop_c).
  • P_push(c): If vld_c remains asserted continuously from some cycle t_o onward while the channel is full (occ_c = B(c)), and the complementary endpoint process continuously requests the matching Pop_c (i.e., rdy_c remains asserted continuously, with the Pop_c boundary request (i.e., BoundaryReq(Pop_c,t) remaining true), then some Pop_c commit must eventually occur (i.e., the full condition is eventually discharged). Equivalently: when both endpoints continuously request the transfer, the system cannot remain stuck forever in the full state without a Pop_c commit.
  • P_pop(c): If rdy_c remains asserted continuously from some cycle t_o onward while the channel is empty (occ_c = 0), and the complementary endpoint process continuously requests the matching Push_c (i.e., vld_c remains asserted continuously with stable payload, with the Push_c boundary request (i.e., BoundaryReq(Push_c,t)) remaining true), then some Push_c commit must eventually occur (i.e., the empty condition is eventually discharged). Equivalently: when

both endpoints continuously request the transfer, the system cannot remain stuck forever in the empty state without a Push_c commit. (For B(c)=0 rendezvous channels: if both endpoints continuously request the transfer from some cycle t_o onward (both requests persistent; guards remain true), then the rendezvous completion eventually occurs.)
These obligations rule out "permanent back-pressure loops" that are logically independent of the local R-rules.

- B5 (System quiescence closure). If at some cycle no observable actions are enabled in any process, then within a bounded number of cycles the system reaches a fixed point and executes no further ε-steps. This prevents a dead, all-disabled state from being hidden behind an infinite tail of internal activity.

These B-rules are assumptions on the execution environment, not guarantees automatically provided by the HLS tool. In practice they place concrete requirements on arbiters, back-pressure, and external masters that interact with the post-HLS RTL.

Priority arbiter example: fair vs unfair priority

Consider a simple two-input priority arbiter inside the post-HLS RTL. Each input is driven by a message-passing channel; the arbiter issues Pop operations to select which request to serve in each cycle.

- Input 0: high-priority channel c_high
- Input 1: low-priority channel c_low

Both channels may have pending requests at the same time.

1. Fair priority arbiter (satisfies B2 / B3).

A fair implementation might still give c_high strict priority in *individual* decisions, but it ensures that a continuously pending c_low request cannot starve. In RTL terms, this can be realized by, for example:

- A round-robin or rotating-priority arbiter that periodically moves the "highest" priority to the next requester, or
- A bounded-burst priority scheme in which c_high may win at most N consecutive cycles while c_low is requesting; after that, the next grant is forced to c_low.

Under these implementations:

- If Pop_low's enable remains true indefinitely (the low-priority channel has a pending request and downstream space is available), the scheduler must eventually grant Pop_low. This satisfies B2 for that action.
- If the low-priority FIFO is full and keeps requesting service, the system ensures that some Pop_low eventually fires, discharging data and freeing space. This is consistent with P_push/P_pop in B3.

Intuitively, the arbiter is still "priority-based," but its micro-architecture ensures that every continuously enabled requester is eventually served.

2. Unfair priority arbiter (violates B2 / B3).

A more naive design might implement:

```
if (req_high) grant_high();
else if (req_low) grant_low();
```

combined with an environment in which req_high can remain asserted indefinitely. In this case, even if req_low is also continuously asserted:

- Pop_low's enabling predicate is true forever, but it is never chosen by the scheduler.
- Low-priority requests can starve indefinitely, even though they are logically ready to fire.

This behavior violates B2 (weak fairness of the scheduler). If c_low's FIFO is full and the only way to make space is to service it, permanent starvation also violates B3's progress expectations for that channel.

In such a design, the safety properties E1–E5 may still hold (trace-equivalence on actions that *do* occur), but the liveness/results that depend on B2/B3—especially the starvation-vs-deadlock arguments in Appendix K—no longer apply.

Patterns that typically satisfy the fairness assumptions

The following design patterns normally satisfy B2 and are compatible with B3/B5, provided the rest of the system is well-behaved:

- Round-robin or rotating-priority arbiters. Any requester that remains enabled will eventually be at the front of the rotation and will be granted.
- Age-based or credit-based arbiters. Requesters accumulate age or credits while waiting; the arbiter prefers older or more-starved requests, guaranteeing that long-pending actions eventually win.
- Bounded-burst fixed priority. Fixed priority is combined with a counter that limits how long a higher-priority requester can dominate while lower-priority requesters are pending. After the burst limit is reached, lower-priority requests are forced to win until the system is "caught up."
- Back-pressure with bounded stall. When ready/valid or similar handshake signals are used, the design (or its environment) must enforce that ready cannot remain low forever while valid remains high, and vice versa. For example:
    - Downstream consumers are required (by specification) to service queues at least once every N cycles when data is present.
    - External bus masters are configured such that they cannot indefinitely defer reading from full DUT FIFOs that are logically part of the interface contract.
- Clock-gating and power-management schemes that respect B5. Once no observable actions are enabled, the design either:
    - Quietly stabilizes (no further internal toggling), or
    - Explicitly enters a low-power state from which it only wakes when some observable action again becomes enabled.

In each case, the intent is the same: continuous enable implies eventual service, and once nothing is enabled, the system converges rather than oscillating internally.

Patterns that tend to violate the fairness assumptions

Conversely, the following patterns are likely to break B2/B3/B5 and therefore fall outside the scope of the liveness arguments in this document:

- Pure fixed-priority arbitration with unbounded high-priority traffic. A static priority tree with no rotation or aging, combined with workloads in which a high-priority master can keep its request asserted indefinitely, can starve lower-priority channels forever.
- "Best-effort" external masters with no progress guarantee. For example:
    - A software driver that reads from a DUT output FIFO only when it happens to poll, and that may be pre-empted indefinitely by higher-priority threads.
    - An external bus or DMA engine that is architecturally allowed to ignore some requesters forever if higher-priority traffic remains heavy.
      Such environments can violate P_push/P_pop by allowing FIFOs to remain full or empty indefinitely while the DUT is continuously requesting progress.
- Circular back-pressure loops with no escape. Two or more processes form a cycle in which each is waiting for the other's channel to change state, and no other process can break the cycle. Without an explicit design-level guarantee that some process in the loop will eventually act (for example, by dropping priority or issuing a compensating Pop/Push), B3 is not satisfied.

- Internal oscillation without quiescence. A scheduler or control FSM that can toggle internal ε-level state forever, even after all observable actions are disabled, violates B5. While such designs are uncommon in practice, patterns involving mis-configured clock gating, asynchronous feedback, or "keep-alive" timers that never expire can have this effect.

In all these cases, the trace-equivalence theorems still describe what happens when actions do occur, but the liveness claims that depend on B2/B3/B5 (for example, "a continuously enabled channel will not starve") are no longer guaranteed. Designers and verification engineers should therefore either:

- Architect their arbiters and environments to satisfy these B-rules, or
- Treat the liveness guarantees in Appendix K as *out of scope* for those particular interfaces, and rely only on the safety-oriented parts of the model.

Appendix O - Support for Multiple Clock Domains

To lift the single-clock formalism of Appendices G–K to a multi-clock setting, we model each clock domain *d* as its own synchronous island with a local cycle counter *clk_d* and apply the existing R- and E-rules unchanged to processes within a fixed domain, interpreting *clk(.)* as *clk_d(.)* for all local synchronization, signal I/O, and message-passing events. All inter-domain communication is required to go through explicit clock-domain-crossing FIFOs; at the abstraction level of Appendix G, each such CDC FIFO is just another bounded channel with capacity *B(c)* and standard ready/valid semantics, so rendezvous pre-HLS channels and buffered post-HLS channels still satisfy FIFO legality (E4), occupancy invariants, and progress obligations *P_push(c)* and *P_pop(c)*, independent of the relative phasing of the two clocks. System behavior and liveness are then expressed in terms of the existing happens-before partial order over observable actions: intra-domain edges are ordered by the local *clk_d*, while each successful Push/Pop pair on a CDC FIFO induces a cross-domain happens-before edge. The weak-fairness and progress assumptions (B2/B3) are strengthened to require fairness of each local scheduler and of each CDC synchronizer, and Appendix K's wait-for-graph and occupancy arguments are applied to this partial order rather than to a single total clock order, so the deadlock-preservation and trace-equivalence properties (E1–E5) continue to hold provided that no raw signals cross clock domains and every cross-domain path uses a CDC FIFO whose implementation is metastability-safe but otherwise abstracted as an ordinary bounded channel.

Appendix P - AI Discussion of Alternative Formal Frameworks

The following links provide an AI discussion of alternative formal frameworks compared to the one presented in this document:

https://drive.google.com/file/d/1ccU-eRW7H2ggh-AZyKnCOzO-KLPejob_/view?usp=sharing

https://chatgpt.com/share/69457119-ec30-8006-8684-9a2a8b19f96f

---

Appendix Q – Multiple Input Pipelines Back-Annotation

Appendix J Remark 2 discusses back-annotation of pipelined loops with a single message passing input.

In more complex cases such as HW pipelines with multiple message-passing inputs, the back-annotation mechanism may elaborate the Sys_B (source-level) simulation by introducing additional internal realization structure. In this document, we adopt the following simplified modeling discipline:

1. Terminology (elaboration and channel accounting). In an elaborated realization, the quantities B(·), occ_·(t), and the E4 availability predicate are interpreted over the explicit abstract channels present in the elaborated model (including any introduced staged/bundle channels). When we continue to write a source-level channel name c at the Σ boundary, we mean the chosen Σ-visible abstract boundary channel associated with c in that elaboration; downstream staging that is reachable only through a join (and therefore may be withheld when "join not met") is represented by distinct internal channels with their own B(·) and occ_·(t), rather than as independent "slack" for c in isolation.
2. All buffering / capacity effects are represented explicitly as finite-capacity message channels (annotated with bounded capacity) that may appear upstream and/or downstream of any internal glue logic; and
3. Any internal "coupler" used to enforce mutual-stall between multiple inputs is purely combinational (stateless) handshake logic.

Concretely, a combinational coupler is a stateless component that (a) reads the upstream channels' valid and data signals and computes/drives the downstream channels' valid and data signals, and (b) reads downstream ready signals and upstream valid signals and then computes/drives upstream ready signals. The coupler contains no internal state and has no separate credit/capacity state; any shared pipeline capacity, per-input staging, skid/elastic buffering, or credit-like effects are modeled only by the explicit finite-capacity channels inserted by the back-annotation realization.

Example (two inputs, staged Pops, II=1, latency=4):
Consider a 4-stage RTL pipeline that (i) performs Pop_c1 in stage 1, (ii) performs Pop_c2 in stage 2, and (iii) performs Push_cout in stage 4, with II=1. To model the pipeline's internal staging and the stage-2 *input coupling* in the source-level Sys_B simulation while keeping the same Σ boundary, the back-annotation elaboration may introduce the following internal realization structure:

- An explicit bounded channel F1 of capacity 1 on the c1 path to represent stage-1 in-flight storage of values that have been accepted from c1 but have not yet reached the stage-2 join point.
- A purely combinational coupler/join at the stage-2 boundary that enforces mutual stall: it allows a transfer to proceed only when (a) a token is available from F1, (b) a token is available from c2, and (c) downstream staging has space; it then presents the paired value(s) to the downstream staging.
- An explicit bounded channel F34 of capacity 2 after the coupler to represent the in-flight capacity of stages 3–4 for the joined/paired work.

This example illustrates why multi-input pipelines cannot always be captured by a single per-channel capacity number in isolation: c1 and c2 do not have independent "slack" because acceptance across the stage-2 join boundary (the paired transfer that consumes a token from c2 together with the staged token from F1) is coupled to availability of the corresponding stage-1 value from c1.
All buffering/capacity resides in the explicit bounded channels (F1, F34, and any additional bounded stages the realization uses); the coupler itself remains stateless combinational handshake logic.

This elaboration is an implementation/refinement of the abstract bounded-FIFO channels at the chosen observable boundary; it does not change the formal interface (Σ) or E4's per-channel FIFO meaning at that boundary. Concretely, any back-annotation elaboration (including extra FIFO stages and/or combinational couplers) must satisfy:

• No new DUT-boundary observables: It introduces no new Σ-observable actions at the chosen DUT boundary, and it does not change the labeling of existing Σ-events (Push_c(v), Pop_c(v), etc.).
• Boundary/E4 preservation: At the chosen observable boundary, each Σ-visible committed Push_c/Pop_c event still denotes a committed transfer on the same abstract channel c, with per-channel FIFO order and the same E4 capacity/enablement interpretation under the annotated B(c) and occ_c(t); any additional internal channels introduced by the elaboration (e.g., staged/bundle channels) are separate abstract channels in the elaborated model with their own B(·), occ_·(t), and E4 availability predicates.
• No double-counting of internal staging: internal transfers within the concrete realization that contributes to the effective capacity (extra FIFO stages, skid/elastic buffering, FIFO→pipeline-stage handoffs, movements across internal staged channels, and value propagation through combinational couplers) are ε-steps and do not create additional Σ-visible committed Push_c/Pop_c events beyond the single boundary Push/Pop events.
• Conservation / projection of internal staging (normative): For each Σ-visible abstract channel c at the chosen observable boundary, the elaborated realization shall admit a fixed projection/accounting map from the occupancies of the explicit internal channels introduced by the elaboration (including staged/bundle/join channels, as applicable) to the boundary occupancy occ_c(t). This projection/accounting map is part of the realization and witnesses that the internal occupancy bookkeeping is only a refinement of the boundary model: (i) every ε-step internal transfer preserves the projected boundary occupancy for each Σ-visible channel, and (ii) each Σ-visible committed Push_c/Pop_c updates the projected boundary occupancy exactly as required by E4 (equivalently, exactly as the boundary occ_c(t) defined from Σ-visible boundary commits). For bundled/joined work items, the projection may assign one internal token to multiple Σ-visible channels as specified by the realization. Therefore, the internal capacities/occupancies B(·), occ_·(t) do not introduce an additional independent Σ-visible state; they only refine the same bounded-FIFO accounting already represented at the chosen boundary.
• Multi-input pipelines / combinational couplers (mutual stall + explicit capacity):
To preserve per-channel E4 at the Σ boundary, any downstream staging that is reachable only through a join (and therefore may be withheld when "join not met") shall be modeled as distinct internal finite-capacity channels with their own B(·) and occ_·(t), rather than being treated as additional independent "slack" for an upstream Σ-visible channel in isolation; the back-annotation elaboration may use combinational couplers to enforce the intended mutual-stall behavior when some required inputs are unavailable. Any "shared capacity" or "distributed staging" effects needed for such a pipeline shall be represented only by explicit finite-capacity channels in the realization (e.g., bounded staging FIFOs, bounded slot/credit channels, bounded bundle channels), not by coupler state. The coupler itself remains stateless combinational handshake logic and does not redefine the per-channel E4 capacity/availability predicates at the chosen observable boundary. Explicit join boundary requirement (normative): The mutual-stall point shall be an explicit staged/bundle/join boundary inside the elaborated realization (i.e., the endpoint of an explicit finite-capacity message channel). "Join not met" shall therefore be expressed as ordinary ChannelDisabled at that explicit internal boundary (because that boundary's own E4 availability predicate fails in the ε-quiescent handshake fixed point). The realization must not force a Σ-visible operation on an abstract channel c at the chosen boundary to be ChannelDisabled solely due to the state of some other channel; any cross-channel dependency shall be expressed only via the explicit staged/bundle channels introduced by the elaboration. Operationally, the coupler is "invisible" at the Σ boundary: it only propagates ready/valid between the explicit staged/bundle channels inside the elaborated realization, and it does not change occ_c(t) (which is defined solely by Σ-visible boundary commits).

• WFG compatibility (Appendix K): The realization must not introduce a new blocking point that is invisible at the abstract boundary. In particular, if the consuming process cannot advance the pending blocking frontier Front_P(σ) (Appendix K.1), then at least one blocking frontier operation (Push/Pop) at the chosen boundary is asserted with a true boundary request and is channel-disabled by the K.1 enablement test; the process is therefore "blocked on message passing" exactly as defined in K.1. Because couplers are purely combinational, they have no internal state transitions; any changes in readiness/validity induced by the coupler are purely a function of (i) downstream readiness and (ii) the same channel occupancies / Σ-visible committed transfers already accounted for by E4/K.1, with "downstream readiness" interpreted in the ε-quiescent (ε-normalized) state used for WFG evaluation. Internal buffer movements across inserted staged channels are ε-steps and are assumed WFG-inert under the same pending-blocking-frontier / Front_P(σ) discipline used elsewhere in Appendix K.
• Combinational well-formedness: The network of combinational couplers and ready/valid connections shall be combinationally acyclic (no pure combinational ready/valid loops). Any required feedback that would otherwise create a combinational loop must pass through an explicit finite-capacity channel stage (and is therefore represented in the back-annotated realization and its effective capacities).

Accordingly, the formal development need not depend on the particular back-annotation construction, provided it respects E4 at the chosen observable boundary and satisfies the constraints above.