5-22-2020

Facilitating Mixed Self-Timed Circuits

Alexandra R. Hanson
Portland State University

Facilitating Mixed Self-Timed Circuits

by
Alexandra Hanson

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of
Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Professor Marly Roncken, M.Sc.

Portland State University

2020

Facilitating Mixed Self-Timed Circuits

Alexandra Hanson
Department of Computer Science
Portland State University
Portland, OR
aleh2@pdx.edu

Abstract—Designers constrain the ordering of computation events in self-timed circuits to ensure the correct behavior of the circuits. Different circuit families utilize different constraints that, when families are combined, may be more difficult to guarantee in combination without inserting delay to postpone necessary events. By analyzing established constraints of different circuit families like Click and GasP, we are able to identify the small changes necessary to either 1) avoid constraints entirely; or 2) decrease the likelihood of necessary delay insertion. Because delay insertion can be tricky for novice designers and because the likelihood of its requirement increases when mixing different self-timed circuit families, we seek to identify simple circuit changes to facilitate the correct mixing of these families.

Index Terms—asynchronous circuits, Click, GasP, NuSMV model-checking, relative timing, self-timed circuits

I. INTRODUCTION

Asynchronous circuits, or self-timed circuits, are circuits that complete their operations independent from a clock signal. Rather than relying on a global signal to coordinate actions like traditional synchronous systems, self-timed circuits utilize local validity signals to indicate operation completion and successful data transfer. Self-timed circuits offer many advantages, such as: low power, low energy, and low electromagnetic radiation, as well as high speed, high delay-tolerance, and high scalability. However, a lack of design tools and insufficient education on asynchronous systems are impediments to their wide adoption, and despite the many benefits of self-timed designs, synchronous circuits are more commonly used in current digital technology [1].

While self-timed circuits do have a high delay tolerance as a benefit, they can still have delay assumptions. These delay assumptions can be validated and implemented using static timing analysis and delay insertion methods similar to those used in traditional synchronous circuits. However, there is a key difference in the nature of the delay assumptions for synchronous versus self-timed systems. In synchronous systems, delay assumptions constrain the ordering of computation events relative to the global clock. In contrast, self-timed systems lack such a global clock, and their delay assumptions thus constrain the ordering of computation events relative to each other. In this thesis, we will model delay assumptions in self-timed systems using the theory of Relative Timing or RT [2]–[4].

Different self-timed circuit families utilize different RT constraints. We have found that it is generally much easier to satisfy RT constraints by the delays of the gates already present in the design when the design is based on a single circuit family. When different self-timed circuit families are mixed in the same design, it is more likely that the combination of such different RT constraints can be met only by inserting extra delay. This makes mixing self-timed circuits more challenging and impedes exchange and reuse of self-timed solutions between different design groups.

This thesis seeks to facilitate mixing, matching, and reuse of self-timed solutions by reducing the likelihood of delay insertion. Specifically, this thesis seeks to analyze the established constraints of the asynchronous circuit families Click [5] and GasP [6], in the context of Roncken's Link and Joint Model [7]-[9]. While previous models for asynchronous design placed the bulk of a system's work onto a single component, the Link and Joint Model instead assigns computation and flow control responsibilities to the "Joint" components and transportation and communication duties to the "Link" components. This model hides circuit details and unifies many of the existing self-timed circuit families with a single model.

By examining the known constraints of Click and GasP, we are able to identify small design changes necessary to either 1) avoid RT constraints entirely; or 2) decrease the likelihood of delay insertion in practice. Because delay insertion can be tricky for novice designers and because the likelihood of its requirement increases when mixing different self-timed circuit families, this thesis identifies the simple circuit changes that can be made to facilitate and

streamline the correct mixing of these families. The NuSMV model-checker [10] is used to examine the validity and completeness of RT constraints for a given Link or Joint design. We outline our process for modeling and refinement in four distinct sections:

(1) MODELING THE FIFO BUFFER AND ITS COMPONENTS, in which we present and discuss a simple Link-Joint-Link buffer model and describe its components.

(2) MIXED CIRCUIT FAMILIES AND THEIR RELATIVE TIMING (RT) CONSTRAINTS, in which we describe the necessary event orderings for our models given in the Relative Timing formalism.

(3) IMPLEMENTATION OF MODELS AND RT CONSTRAINTS IN NUSMV, in which we present the implementation of our models in the symbolic model-checker NuSMV by example of the Joint component.

(4) MINIMAL CIRCUIT DESIGN CHANGES, in which we propose and discuss the small circuit changes that minimize the amount of delay insertion required.

The NuSMV model-checker code is contained in the attached appendixes A, B, and C. Appendix A contains the NuSMV implementation and results of the mixed FIFO model of section II. Appendixes B and C contain the NuSMV implementation and results of two minimal circuit changes that we propose in section V.

II. MODELING THE FIFO BUFFER AND ITS COMPONENTS

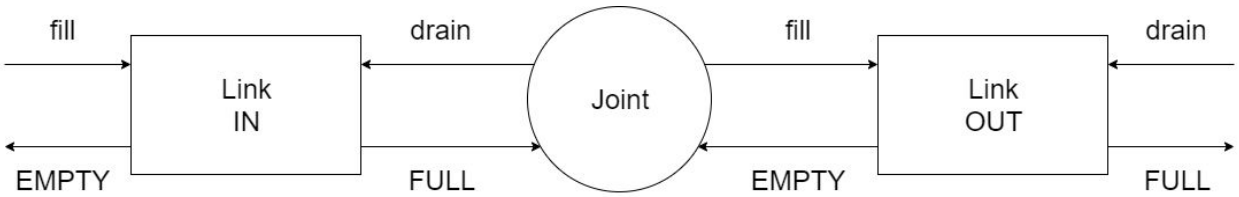

Fig. 1: A First-In-First-Out (FIFO) Buffer, LinkIN-Joint-LinkOUT.

The simple FIFO buffer in Fig. 1 illustrates a high-level dataflow pipeline for the Link and Joint Model. In this example, we have components LinkIN, Joint, and LinkOUT, with Boolean interface signals *fill*, *drain*, FULL, and EMPTY [7]-[9].

fill:	an active high (1/true) signal issued by a Joint to communicate to a receiving Link that the Joint has newly computed data for the Link.
drain:	an active high (1/true) signal issued by a Joint to communicate to a sending Link that the Joint no longer needs the Link's data.
FULL:	an active high (1/true) signal issued by a Link to a receiving Joint that the Link has valid data for the next computation by the Joint.
EMPTY:	an active high (1/true) signal issued by a Link to a sending Joint that the Link is ready to receive new data.

Note: Joints fill only their EMPTY receiving Links and drain only their FULL sending Links.

In Fig. 1, when LinkIn is filled with data it declares itself FULL by raising its FULL signal. Likewise, when LinkOUT is drained and thus its data are no longer useful, it declares itself EMPTY by raising its EMPTY signal. When the Joint sees that LinkIn is FULL and LinkOUT is EMPTY, it performs its computation on the data and simultaneously *drains* LinkIN and *fills* LinkOUT by raising its *drain* signal to LinkIN and raising its *fill* signal to LinkOUT. LinkOUT accepts the computed data, and will now simultaneously output a low EMPTY signal to the sending Joint and a high FULL signal to the (implied) receiving Joint. Likewise, LinkIN will simultaneously output a low FULL signal to the receiving Joint and a high EMPTY signal to the (implied) sending Joint. When the Joint sees that LinkIN is no longer FULL and LinkOUT is no longer EMPTY, it stops the computation and its *fill* and *drain* actions by lowering its *drain* signal to the LinkIN and by lowering its *fill* signal to LinkOUT.

Although more generally a Joint can have zero or more input Links and zero or more output Links, we use this simple FIFO example for simplicity and understandability. Modeling the high-level behavior of this FIFO in NuSMV requires modeling distinct Link and Joint components before we model them in combination. Because we seek to demonstrate that the

different Link control circuitries of the Click and GasP circuit families are interchangeable, our models for these Link components reflect both the distinct implementations of these families as well as their overall identical function as Links. A discussion of the circuit models for the Joint, Click Link, and GasP Link follows in subsections II-A, II-B, and II-C respectively. Click and GasP use similar circuits for data storage and computation. Because this study focuses on where they differ, Fig. 1 ignores the data signals as does the rest of this thesis.

II-A. The Joint Model

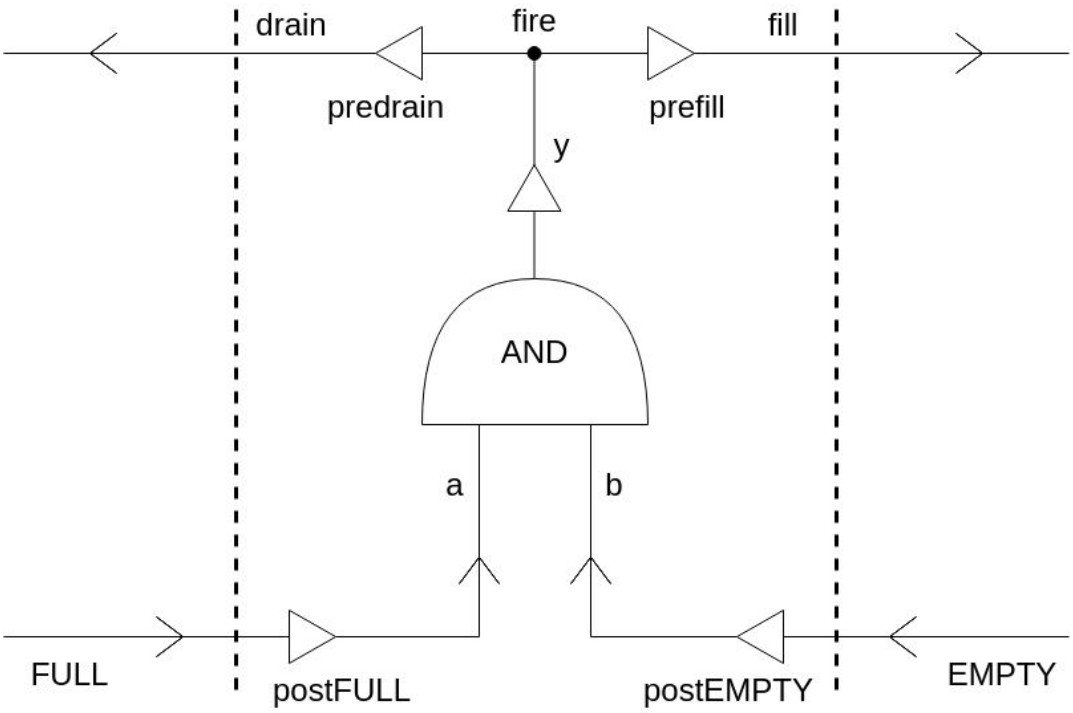

Fig. 2: Joint Model.

The Joint circuit model is pictured in Fig. 2. It consists of the following components:

- Two input signals FULL and EMPTY with respective associated buffers postFULL and postEMPTY.
- An AND gate to perform the logical conjunction of the two inputs *a* (corresponding to signal FULL) and *b* (corresponding to signal EMPTY).
- An output of the AND gate that propagates through buffer *y* to signal *fire*, and from there through buffers *prefill* and *predrain* to output signals *fill* and *drain*, respectively.

Note: The buffers in Fig. 2 serve to model wire and amplification delays.

The FULL and EMPTY signals come from an implied LinkIN and LinkOUT (see Fig. 1). The two dotted lines in Fig. 2 represent the interface between these implied Link components and the Joint. When both EMPTY and FULL signals are high, both *fill* and *drain* will go high. When either or both EMPTY and FULL are low, *fill* and *drain* will go low.

II-B. The Click Link Model

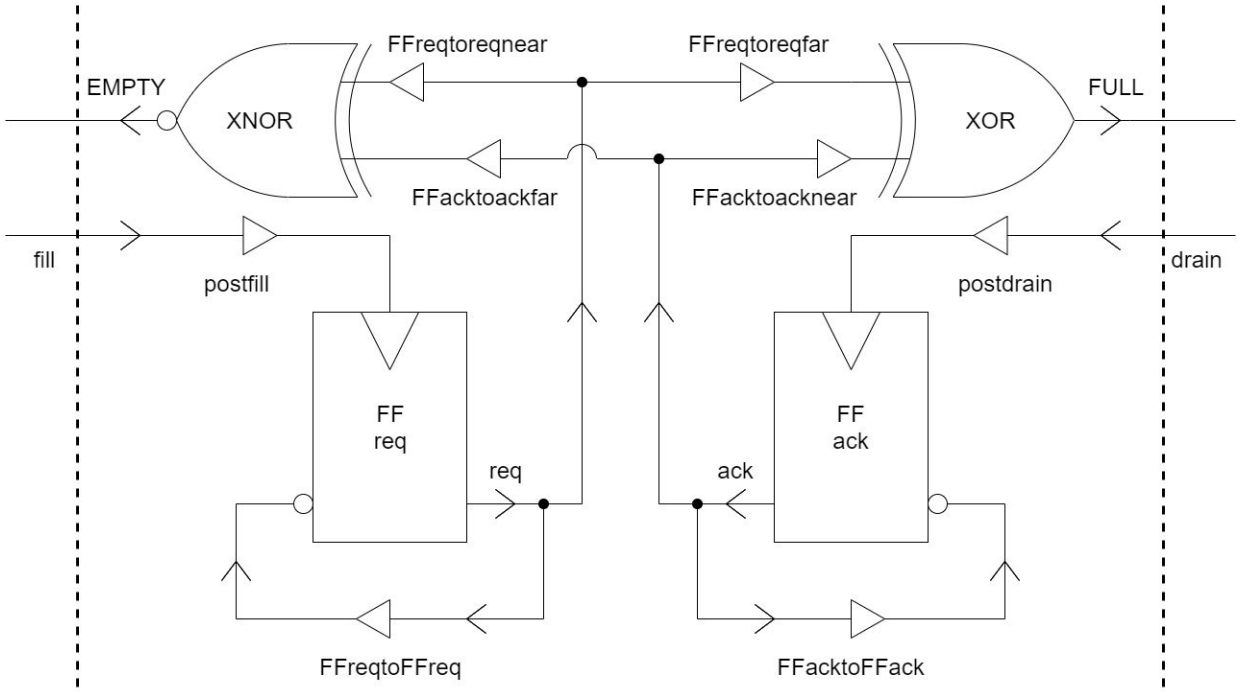

Fig. 3: Click Link Model.

The Click Link circuit model is pictured in Fig. 3. It consists of the following components:

- Two input signals *fill* and *drain* with respective associated buffers *postfill* and *postdrain*.
- Two flip-flops *FFreq* and *FFack* to generate and store the state of the Link. Additional buffer *FFreqtoFFreq* is associated with flip-flop *FFreq* while buffer *FFacktoFFack* is associated with flip-flop *FFack*.
- An XNOR gate taking as inputs the output of flip-flop *FFreq* through buffer *FFreqtoreqnear* and the output of flip-flop *FFack* through buffer *FFacktoackfar*.
- An XOR gate taking as inputs the output of flip-flop *FFreq* thorough buffer *FFreqtoreqfar* and the output of flip-flop *FFack* through buffer *FFacktoacknear*.
- The output signal EMPTY associated with the XNOR gate and the output signal FULL associated with the XOR gate.

Note: As before in Fig. 2, the buffers model wire and amplification delays.

The *fill* and *drain* signals come from implied sending and receiving Joints; the two dotted lines in Fig. 3 represent the interface between the implied Joint components and the Click Link. The behavior of the two flip-flop components *FFreq* and *FFack* are identical. Each flip-flop receives two inputs: an enabling input *fill* or *drain* and an inversion of the given flip-flop's previous output. Upon a rising edge of the enabling input, the output of the flip-flop will flip. The new output is also used as an input to both the XNOR and XOR gate.

FFreq and *FFack* encode a two-signal state, typical of Click. *FFreq* controls signal *req* ("request"). *FFack* controls signal *ack* ("acknowledge"). Fig. 3 encodes EMPTY as "*req* = *ack*" or XNOR(req, ack), and FULL as "*req* != *ack*" or XOR(req, ack). Thus when *req* and *ack* are both low or both high, the Click Link is EMPTY, and otherwise the Click Link is FULL. Specifically, an enabling input signal *fill* (on an EMPTY Click Link) to flip-flop *FFreq* will produce an output value that differs from the output of *FFack*, while an enabling input signal *drain* (on a FULL Click Link) to flip-flop *FFack* will produce an output value that is the same as the output of *FFreq*. The XNOR and XOR gates generate the FULL or EMPTY state of the Link by determining whether or not the *req* and *ack* associated inputs to these gates differ. If the *req* and *ack* input signals are the same (both high or both low), then the XNOR gate will output a high EMPTY signal and the XOR gate will output a low FULL signal—the Link is EMPTY. In contrast, if the *req* and *ack* input signals differ (one is high and one is low), then the XNOR gate will output a low EMPTY signal and the XOR gate will output a high FULL signal—the Link is FULL.

II-C. The GasP Link Model

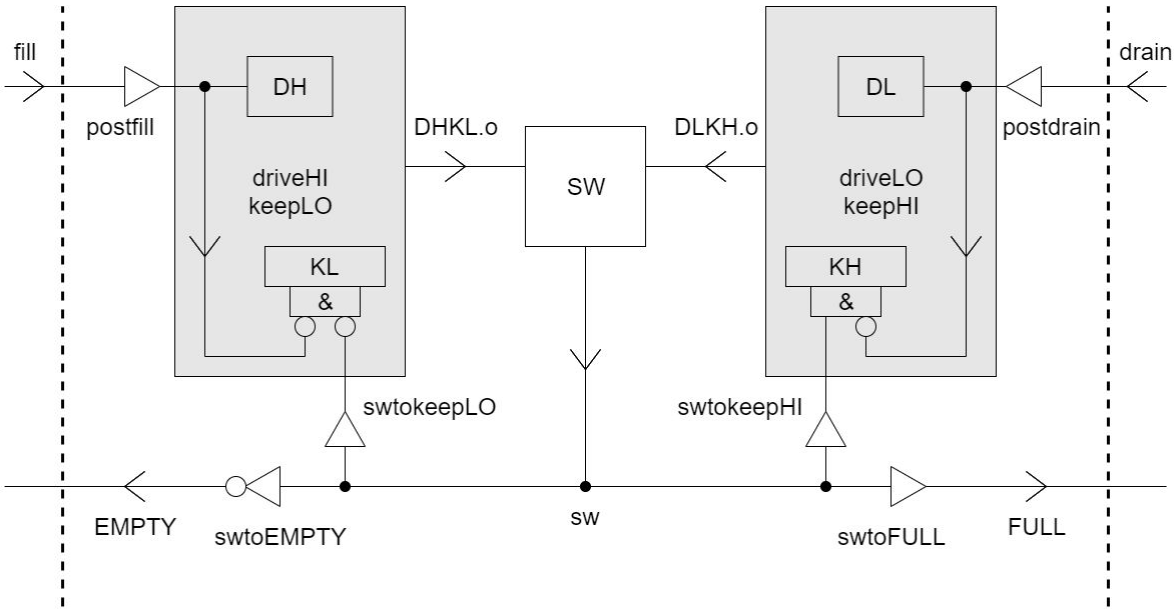

Fig. 4: GasP Link Model.

The GasP Link model is pictured in Fig. 4. It consists of the following components:

- Two input signals *fill* and *drain* with respective associated buffers *postfill* and *postdrain*.
- A module *driveHIkeepLO* that abstracts out the CMOS logic circuit responsible for driving the state of the Link high (FULL) and keeping the state of the Link low (EMPTY).
- A module *driveLOkeepHI* that abstracts out the CMOS logic circuit responsible for driving the state of the Link low (EMPTY) and keeping the state of the Link high (FULL).
- Two buffers *swtokeepLO* and *swtokeepHI* which serve as inputs to respective modules *driveHIkeepLO* and *driveLOkeepHI*.
- A module SW ("statewire") responsible for interpreting the outputs of the *driveHIkeepLO* and *driveLOkeepHI* modules .
- The output signal EMPTY associated with inverter *swtoEMPTY* and the output signal FULL associated with buffer *swtoFULL*.

Note: As before, in Fig. 2 and Fig. 3, the buffers model wire and amplification delays.

Like the Click Model, the *fill* and *drain* signals of the GasP Model come from implied sending and receiving Joints, and the two dotted lines in Fig. 4 represent the interface between the implied Joint components and the GasP Link. Modules *driveHIkeepLO* and *driveLOkeepHI* simplify the necessary CMOS logic into two distinct components which determine the current state of the Link. The input *fill* to module *driveHIkeepLO* drives the state of the Link high, while the combination of inverted inputs !*fill* and !*swtokeepLO* keep the state of the Link low. The input *drain* to module *driveLOkeepHI* drives the state of the Link low, while the combination of inverted input !*drain* and signal *swtokeepHI* keep the state of the Link high.

GasP uses a one-signal state. Fig. 4 encodes EMPTY as "!SW" or SW is low, and FULL as "SW" or SW is high. Specifically, module SW generates the current state of the Link (FULL or EMPTY) by interpreting its inputs from modules *driveHIkeepLO* and *driveLOkeepHI*. SW is also responsible for monitoring correct communication between modules *driveHIkeepLO* and *driveLOkeepHI* by checking whether their outputs conflict or "float", i.e., neither module drives the state of the Link.

The output of SW propagates to buffers *swtoFULL*, *swtokeepHI*, *swtokeepLO*, and inverter *swtoEMPTY*. If SW generates a high signal, buffer *swtoFULL* outputs a high FULL signal and buffer *swtokeepHI* outputs a high signal to keep the state of the Link FULL; simultaneously, inverter *swtoEMPTY* outputs a low EMPTY signal and the high signal of buffer *swtokeepLO* is also inverted to stop the keepLO circuitry from keeping the state EMPTY. Likewise, if SW generates a low signal, inverter *swtoEMPTY* outputs a high EMPTY signal and the low signal of buffer *swtokeepLO* is also inverted so that the state of the Link is kept EMPTY.

Additionally, buffer *swtoFULL* outputs a low FULL signal and buffer *swtokeepHI* outputs a low signal to stop the keepHI circuitry from keeping the state FULL.

III. MIXED CIRCUIT FAMILIES AND THEIR RELATIVE TIMING (RT) CONSTRAINTS

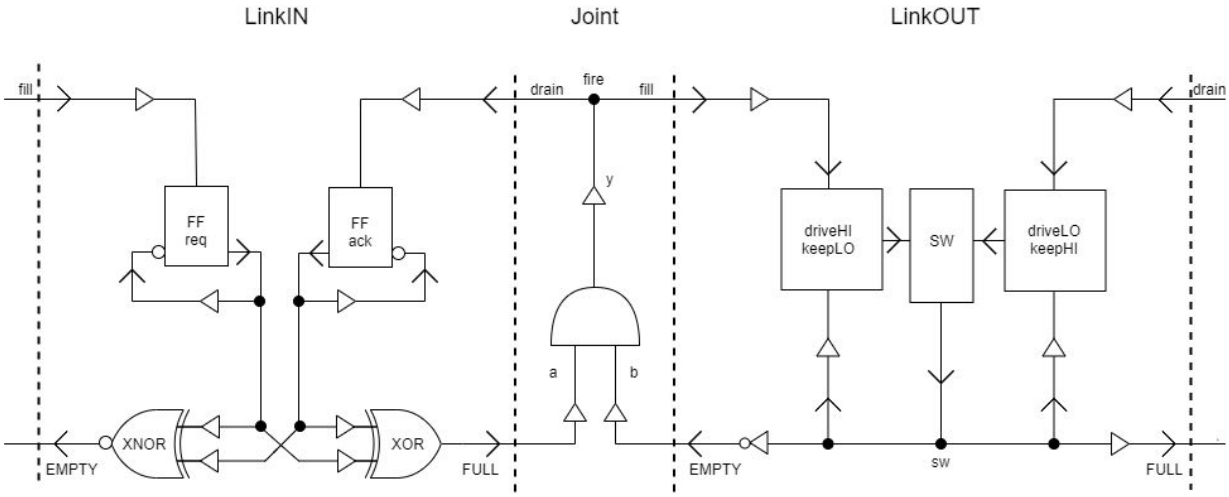

Fig. 5: The mixed Family FIFO model "Click Link - Joint - GasP Link."

The mixed family FIFO model pictured in Fig. 5 is a combination of our Joint, Click Link, and GasP Link models. Fig. 5 lacks the buffers *prefill* and *predrain* present in Fig. 2. Their functions, representing the individual wire and amplification delays in signals *fill* and *drain*, are already included in Fig. 5 with the buffers *postfill* in Click Link (see Fig. 3) and *postdrain* in the GasP Link (see Fig. 4). The Link and Joint components require unique timing constraints to ensure correct behavior. Relative Timing (RT) Constraints [2]-[3] are explicit event orderings not enforced by the design but required to ensure correct behavior. Not enforcing these constraints by design often simplifies the circuit implementation of a component. The RT constraints that follow in this thesis are of the form:

$$e_0 \rightarrow e_1 < e_2$$

This can be read as "if event e_0 happens, then event e_1 must happen before event e_2." Given this constraint, it will be necessary to postpone event e_2 in the real circuit if it occurs between events e_0 and e_1 by inserting physical delay. Event e_0 is called the *point of divergence* (POD), e_1 the *early event*, and e_2 the *late event*.

Our work with RT constraints for the mixed family FIFO model extends the work of Hoon Park [4], [11]. Park determined the RT constraints of asynchronous Click circuits prior to the development of the Link and Joint Model [7] and tied the verified constraints to Static Timing Analysis code in the ARCwelder compiler. By applying Park's work and methodology to Link and Joint components, we can understand the consequences of mixing different Link control circuits and ultimately identify simple circuit changes that can be made to ease the challenge of delay insertion for the Link and Joint Model.

III-A. RT Constraints associated with the Joint

The Joint component's RT constraints ensure that signals reset appropriately to avoid duplicate or missed computations. We prevent the FULL and EMPTY signals at the Joint's interface from becoming "mismatched." For example, a newly high EMPTY signal cannot be paired with the previous FULL signal as this would result in a duplicate computation. Similarly, a newly high FULL signal cannot be paired with the previous EMPTY signal as this would result in a missed computation. Two constraints are assigned to the Joint to relate the events of LinkIN to those of LinkOUT and to prevent erroneous behavior.

Note: See Fig. 2 for the signal names associated with the following RT constraints.

Joint-RT1:

We would like to "reset" an action only when the Joint has observed that LinkIN and LinkOUT have both successfully fulfilled their minimally necessary state storage and transportation duties, namely updating their state information. We require that once signal *fire* has gone high, signals FULL and EMPTY will both go low before signal *fire* can go low again. This is expressed by the following RT constraint:

$$\text{fire+} \rightarrow \{\text{EMPTY-}, \text{FULL-}\} < \text{fire-}$$

Joint-RT2:

We would like to prevent new inputs related to FULL+ and EMPTY+ until the Joint's current action has properly reset. We require that once signal *fire* has gone high, signals *a*, *b*, *fire*, *fill*, and *drain*, as well as their derived signals into the Links, will all go low before signals *a* and *b* can go high again. By causality this can be simplified to *a, b,* and Link contributions from *postfill* and *postdrain* go low before either *a* or *b* can go high again. See Fig. 3 and Fig. 4 for buffer signals *postfill* and *postdrain*. The RT constraint is:

$$\text{fire+} \rightarrow \{\text{a-}, \text{b-}, \text{LinkIN.postdrain-}, \text{LinkOUT.postfill-}\} < \text{a+}, \text{b+}$$

III-B. RT constraints associated with the Click LinkIN

Firstly, the RT constraints of the Click Link component ensure that its internal flip-flops perform their flipping action only once per enabling input.

Note: see Fig. 3 for the signal names associated with the following RT constraints.

Click-RT1:

We require that if the output of flip-flop *FFreq* goes high (or low), then the related buffer *FFreqtoFFreq* must go high (or low) before the flip-flop's enabling input, *postfill*, can go high again.

a. If *FFreq.Q* goes high, then buffer *FFreqtoFFreq* must go high before the buffer *postfill* can go high again.

FFreq.Q+ → FFreqtoFFreq+ < postfill+

b. If *FFreq.Q* goes low, then buffer *FFreqtoFFreq* must go low before the buffer *postfill* can go high again.

FFreq.Q- → FFreqtoFFreq- < postfill+

Click-RT2:

We require that if the output of the flip-flop *FFack* goes high (or low), then the related buffer *FFacktoFFack* must go high (or low) before the flip-flop's enabling input, *postdrain*, can go high again.

a. If *FFack.Q* goes high, then buffer *FFacktoFFack* must go high before the buffer *postdrain* can go high again.

FFack.Q+ → FFacktoFFack+ < postdrain+

b. If *FFack.Q* goes low, then buffer *FFacktoFFack* must go low before the buffer *postdrain* can go high again.

FFack.Q- → FFacktoFFack- < postdrain+

Additional RT constraints of the Click Link component ensure that there is no conflict over the FULL or EMPTY state of the Link. To discuss the prevention of a Link state conflict, we distinguish the two ends of the Link as *near-end* and *far-end* in relation to the Joint that began the current Link operation. We specify that the near-end must complete its task before the far-end can begin its own; for example, a Link must be fully *drained* before it can be *filled* again. This avoids a conflict between the two ends because it prevents them from entering a logical fight over an XOR or XNOR gate where both ends drive the inputs at the same time.

Note: See Fig. 2 for the signal names *a* and *b* associated with the following RT constraints.

Click-RT3:

We would like the near-end action to be performed before the far-end response arrives at the XOR or XNOR gates.

a. We require that a high *fill* signal will propagate sufficiently far into an implied JointIn as a low EMPTY signal through the XNOR gate before a *drain* signal arrives at the XNOR to drive the EMPTY signal high again. Specifically, if the *fill* signal goes high, signal *b* of some implied JointIn must go low before buffer *FFacktoackfar* can go high.

fill+ → impliedJointIn.b- < FFacktoackfar+

b. We require that a high *drain* signal will propagate far enough into the Joint as a low FULL signal through the XOR gate before a *fill* signal arrives at the XOR gate to drive the FULL signal high again. Specifically, if the *drain* signal goes high, signal *a* of the Joint must go low before buffer *FFreqtoreqfar* can go high.

drain+ → Joint.a- < FFreqtoreqfar+

III-C. RT constraints associated with the GasP LinkOUT

The first two RT constraints of the GasP Link component ensure that the appropriate FULL or EMPTY state is maintained. This is guaranteed by activating the far-end keep signal and deactivating the near-end keep signal before the current driving signal goes low.

Note: see Fig. 4 for the signal names associated with the following RT constraints.

GasP-RT1:

We require that the far-end keepers are activated sufficiently soon to maintain the appropriate FULL or EMPTY state.

a. If the *fill* signal goes high, buffer *swtokeepHI* must go high before buffer *postfill* can go low.

fill+ → swtokeepHI+ < postfill-

b. If the *drain* signal goes high, buffer *swtokeepLO* must go low before buffer *postdrain* can go low.

drain+ → swtokeepLO- < postdrain-

GasP-RT2:

We require that the near-end keepers are deactivated to avoid a fight between near- and far-end keepers.

a. If the *fill* signal goes high, buffer *swtokeepLO* must go high before buffer *postfill* can go low.

fill+ → swtokeepLO+ < postfill-

b. If the *drain* signal goes high, buffer *swtokeepHI* must go low before buffer *postdrain* can go low.

drain+ → swtokeepHI- < postdrain-

Like the Click LinkIN, the additional RT constraints of the GasP Link component ensure that there is no conflict over the FULL or EMPTY state of the Link. Two related constraints guarantee that modules *driveHIkeepLO* and *driveLOkeepHI* do not simultaneously attempt to drive the state of the Link FULL or EMPTY respectively. Again, like the previous Click-RT3 example, a Link must be fully *drained* before it can be *filled* again and vice versa.

GasP-RT3:

We would like for the current *driveHIkeepLO* or *driveLOkeepHI* module to propagate its signal far enough into the Joint and stop driving before the opposite *driveLOkeepHI* or *driveHIkeepLO* module starts driving.

a. Once the *fill* signal goes high, both buffer *postfill* and signal *b* of the Joint must go low before buffer *postdrain* can go high.

   fill+ → {postfill-, Joint.b-} < postdrain+

b. Once the *drain* signal goes high, both buffer *postdrain* and signal *a* of some implied JointOut must go low before buffer *postfill* can go high.

   drain+ → {postdrain-, impliedJointOut.a-} < postfill+

IV. IMPLEMENTATION OF MODELS AND RT CONSTRAINTS IN NUSMV

The mixed family FIFO model, along with its associated relative timing constraints, has been implemented and formally verified using the symbolic model checker NuSMV. Because our circuit models are finite state systems and can be described in propositional calculus, NuSMV is an effective choice of model checker [10], [12]. For our purposes, we have used NuSMV version 2.5.4, the same version used by Hoon Park et al. [4], [11]. NuSMV's model of execution requires a formal description of our circuit and runtime properties that we want NuSMV to verify for all circuit executions.

The formal circuit description consists of gate definitions, connections between these gates, and the circuit's relative timing constraints. Because the behavior of our mixed family FIFO model has simple flow control, we consider *semimodularity* properties sufficient to verify this behavior. Here, semimodularity means that an enabled signal change must occur before it becomes disabled [13]. We apply Park's updated definition of this paradigm, which considers semimodularity in the context of RT constraints, where "enabled" implies that none of the RT constraints block the signal change. Our circuit is correct if it is semimodular. Semimodularity checks are built into our NuSMV gate models such that the properties are either achieved in the circuit by design or associated with an RT constraint. If a gate model fails a semimodularity check, NuSMV generates an appropriate counterexample that can be remediated with a new RT constraint.

Our model is run with an interleaving semantics, which models parallel execution by interleaving individual actions. In this execution model, time is partitioned in steps and execution occurs at the gate level, where at most one gate is selected per step. Upon selecting a gate, the gate will either (1) be unready and unable to change because the gate's output already matches what the input requests; (2) be ready to change, but unable to do so because the gate is blocked by an RT constraint; or (3) be ready and able to change, in which case its output signal will change in this step. Fairness conditions [12] are introduced to ensure that each gate will be selected within a finite number of steps at each stage in the execution.

In the following subsections, we will present and discuss the NuSMV implementation of the Joint model and its associated relative timing constraints. See Appendix A for the NuSMV implementations of Click and GasP, as well as their associated RT constraints.

IV-A. NuSMV implementation for the Joint Model

The Joint model has been implemented in NuSMV via a module declaration to encapsulate its components and constraints. This implementation is included below. Note how closely it follows the Joint in Fig. 5.

```
MODULE Joint (FULL, EMPTY, stopfall_fire, stoprise_andin)
    VAR
        buf_postfull       : process cgate (FULL,  FALSE,
```

```
                                    stoprise_andin, FALSE);
        buf_postempty      : process cgate (EMPTY, FALSE,
                                    stoprise_andin, FALSE);
        and                : process cgate (andin1 & andin2, FALSE,
                                    FALSE, FALSE);
        buf_postand        : process cgate (and.val, FALSE, FALSE,
                                    stopfall_fire);
    DEFINE
        andin1:= buf_postfull.val;
        andin2:= buf_postempty.val;
        fire  := buf_postand.val;
        fill  := fire;
        drain := fire;
    --PROPERTIES
    FAIRNESS running
--END MODULE Joint
```

Module Joint is instantiated with inputs FULL, EMPTY, *stopfall_fire*, and *stoprise_andin*. Its instantiation within Module main will be discussed in Section IV-B.

In the above implementation,

FULL: a Boolean from the input Link (LinkIN)
EMPTY: a Boolean from the output Link (LinkOUT)
fill: a Boolean to the output Link (LinkOUT)
drain: a Boolean to the input Link (LinkIN)

The *stopfall_fire* and *stoprise_andin* are Boolean imported RT constraint stops. These constraints will be discussed at length in Section IV-B.

Under the VAR declaration, state variables *buf_postfull*, *buf_postempty*, *and*, and *buf_postand* are each declared as an additional instantiated module *cgate*. These state variables correspond respectively to the buffers postFULL, postEMPTY, the AND gate, and the *y* buffer as seen in the abstract circuit model of Fig. 5. In Fig. 5, the need to distinguish *drain* from *fill* is already achieved by the two inverters present in the *drain* and *fill* signals in LinkIN and LinkOUT. The *cgate* module is a model for any combinational gate. It produces an output *val*, and takes as inputs:

set: a Boolean function specifying the desired output of the gate.
init_val: an initial Boolean value for val.
stoprise: a Boolean function indicating whether or not (a low) *val* is permitted to rise. This is an RT constraint that blocks a low to high transition of output

val. The Joint module uses *stoprise_andin* to block low to high output transitions of buffers postFULL and postEMPTY.

stopfall: a Boolean function indicating whether or not (a high) *val* is permitted to fall. This is an RT constraint that blocks a high to low transition of output *val*. The Joint module uses *stopfall_fire* to block high to low output transitions of gate *buf_postand*.

Note: Please see Appendix A1 for the implementation of module *cgate*.

In the DEFINE declaration of module Joint, concise identifiers are associated with expressions that we will use in the RT constraints, which follow in Section IV-B. The interest points *a* and *b* of Fig. 5 are identified as *andin1* and *andin2*, while the other relevant signals and interest points retain the same names: *fire*, *fill*, and *drain*. Interest point *andin1* is assigned the output *val* of buffer *buf_postfull*, while interest point *andin2* is assigned the output *val* of buffer *buf_postempty*. Interest point *fire* is assigned the output *val* of buffer *buf_postand*, and the signals *fill* and *drain* are assigned the value of *fire*.

Finally, fairness conditions are verified with the constraint `FAIRNESS running`, which ensures that each gate will be selected for execution within a finite number of steps.

IV-B. NuSMV implementation of Joint RT Constraints

Module Joint is instantiated in the Module main as thisJoint (see Appendix A2 for further detail). It's instantiated with signal FULL from LinkIN, signal EMPTY from LinkOUT, and the *stopfall_fire* and *stoprise_andin* RT constraints that are defined within main.

```
thisJoint : process Joint  (FULL, EMPTY, stopfall_fire,
                                  stoprise_andin);
```

As can be seen in the previous Section IV-A, buffer *buf_postand* utilizes the *stopfall_fire* constraint to indicate whether its *val* output is permitted to transition from a high to low signal (i.e. stopping its fall). The Boolean stop *stopfall_fire* implements the RT constraint Joint-RT1 found in Section III-A. The NuSMV definition of the *stopfall_fire* constraint follows and reads as "After *fire* rises, both EMPTY and FULL must be low before *fire* may fall."

```
VAR
    -- fire+ -> EMPTY-, FULL- < fire-
        Joint-RT1a : process rt (fire, !EMPTY, GREEN);
        Joint-RT1b : process rt (fire, !FULL, GREEN);
DEFINE
        stopfall_fire := Joint-RT1a.stop | Joint-RT1b.stop;
```

This definition of *stopfall_fire*, like the definition for the *stroprise_andin* constraint shown further on, declares an additional instantiated module *rt*. The *rt* module, included in Appendix A1, encapsulates the logic necessary to implement relative timing constraints on given events. Module *rt* contains an internal state variable *stop*, responsible for preventing a late event via the formal *stop_rise* or *stop_fall* parameters associated with relevant modules. When true, *stop_rise* prevents a rising output transition of the constrained gate. Likewise, when true, *stop_fall* prevents a falling transition of the constrained gate. We use *stop_rise* and *stop_fall* as input parameters of the gate that produces the late event. The inputs of *rt* are:

eventPOD: the Boolean function specifying the point of divergence for the constraint
eventEARLY: the Boolean function specifying the early event that releases the constraint
init_rt: the initialization state for the internal state variable *stoplight*

A process *rt* is instantiated as `process rt (eventPOD, eventEARLY, init_rt).` In constraint *Joint-RT1a*, the point of divergence is a rising *fire* signal, the early event releasing the constraint is a falling EMPTY signal, and the initialization state for *stoplight* is GREEN (i.e., *Joint-RT1a.stop* is false). Similarly, in constraint *Joint-RT1b*, the point of divergence is a rising *fire* signal, the early event releasing the constraint is a falling FULL signal, and the initialization state for *stoplight* is also GREEN. Because the *stopfall_fire* constraint consists of a disjunction of *Joint-RT1a* and *Joint-RT1b*, the *stop* state variables of *Joint-RT1a* and *Joint-RT1b* must both be false for *stopfall_fire* to be false so as to allow the late event—here, a high to low transition of signal *fire* during execution.

Buffers *buf_postempty* and *buf_postfull* of the Joint module (see Section IV-A) utilize the *stroprise_andin* constraint to indicate whether their respective *val* outputs are permitted to transition from a low to high signal (i.e., stopping their rise). The Boolean stop *stoprise_andin* implements the RT constraint Joint-RT2 found in Section III-A. The NuSMV definition of the *stoprise_andin* constraint is as follows:

```
VAR
    -- fire+ ->
    -- {andin1-, andin2-, LinkIN.postdrain-, LinkOUT.postfill-}
    -- < andin1+, andin2+
    Joint-RT2a: process rt (fire, !thisJoint.andin1, GREEN);
    Joint-RT2b: process rt (fire, !thisJoint.andin2, GREEN);
    Joint-RT2c: process rt (fire, !LinkIn.postdrain, GREEN);
    Joint-RT2d: process rt (fire, !LinkOut.postfill, GREEN);
DEFINE
    stoprise_andin :=
    Joint-RT2a.stop | Joint-RT2b.stop | Joint-RT2c.stop |
    Joint-RT2d.stop;
```

The *stoprise_andin* constraint consists of the four RT constraints *Joint-RT2a*, *Joint-RT2b*, *Joint-RT2c*, and *Joint-RT2d*, and, like the previous *stopfall_fire* constraint, the *stop* state variables of all four constraints must be false for *stoprise_andin* to be false so as to allow both late events—here, rising outputs of *buf_postfull* and *buf_postempty*.

V. MINIMAL CIRCUIT DESIGN CHANGES

Once individual Link and Joint models and their RT constraints were implemented and verified in NuSMV [10], we created a FIFO with mixed families as in Fig. 5 with its RT constraints as in Section III.

Pictured below in Fig. 6 is the pipeline of our modeling and verification process. We used this approach to identify and introduce simple circuit design changes to either 1) avoid some of the mixed FIFO RT constraints entirely; or 2) minimize the likelihood and amount of required delay to insert into the circuit when implemented on silicon or otherwise.

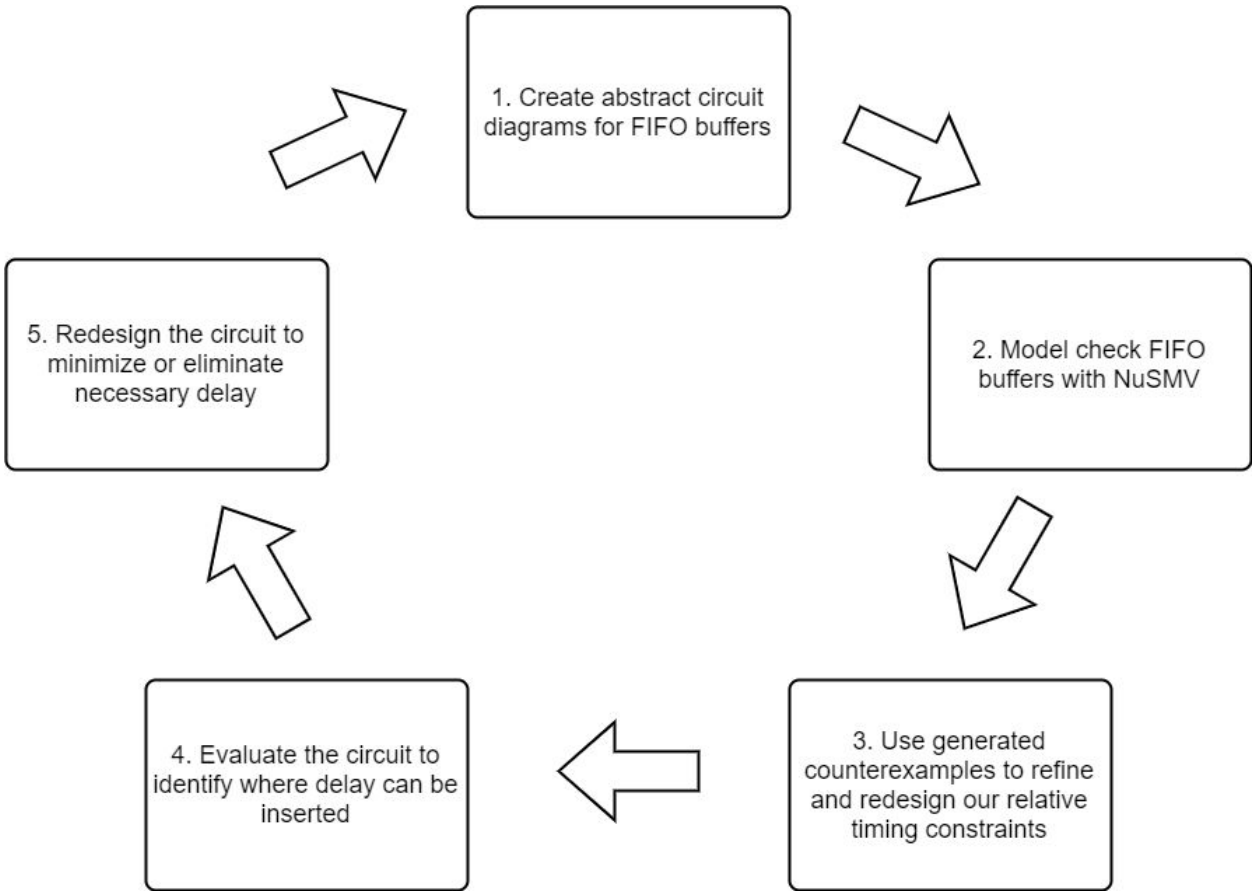

Fig. 6: Modeling and verification approach.

As presented in Sections II through IV, we created abstract circuit diagrams for the FIFO buffer and implemented our models in NuSMV (Steps 1 and 2 in Fig. 6). Model-checking the FIFO buffers generated counterexamples as in Step 3 in Fig. 6. This step required the following additional cycle: (a) Run NuSMV; (b) As long as there are counterexamples, pick a counterexample and generate an RT constraint to address it such that the counterexample no

longer exists; and (c) Add the new RT constraint to the NuSMV model. We remained on Step 3. above and looped through steps (a) through (c) until (a) no longer generated counterexamples.

Once we successfully verified the RT constraints presented in Section III for the mixed FIFO buffer, such that these constraints no longer generated counterexamples, we moved on to Step 4 in Fig. 6. After evaluating the circuit to identify where delay could be inserted, we selected the RT constraint which most restricted the circuit in order to redesign or eliminate the necessary delay insertion (Step 5 in Fig. 6). From here, we made adjustments to our abstract circuit diagrams as appropriate and began the Fig. 6 cycle again until we were satisfied with our redesigns.

This thesis investigates two adjustments to the model in particular: one specific to the GasP Link and the other specific to the Joint. Both of these modifications have broader implications than the simple mixed family FIFO model discussed in this work. The adjustment to the GasP Link component applies to multiple circuit families that work with latch-based designs rather than flip-flops—i.e., circuit families that are similar to GasP and unlike Click. The Joint adjustment applies to other Joints, with more complex flow control than the First-In-First-Out control flow of the Joint example presented in this thesis. The GasP Link and Joint redesigns, as well as the motivations behind them, are described in Section V-A and Section V-B respectively.

V-A. Making the GasP Link Edge-Triggered

The adjustment to the GasP Link was motivated by a comparison of RT constraints Click-RT1 and Click-RT2 to constraints GasP-RT1 and GasP-RT2. Both of these different RT constraint sets have a similar purpose: maintain the appropriate FULL or EMPTY Link state. However, Click-RT1 and Click-RT2 are dependent on rising late signals from *postfill* and *postdrain*, reflecting the edge-triggered nature of the flip-flops that store the Click Link state. In contrast, GasP-RT1 and GasP-RT2 are dependent on falling late signals from *postfill* and *postdrain*, reflecting the level-triggered nature of the *DriveHIkeepLO* and *DriveLOkeepHI* modules that represent the latches storing the GasP Link state.

The *fill* signal that begins each Link action by going high and ends each Link action by going low will ultimately cause each *postfill* and *postdrain* that it steers to first rise and then fall—and then rise and fall again at the next action. GasP-RT1 and GasP-RT2 must be satisfied by the time *postfill* and *postdrain* fall, while the similar Click constraints Click-RT1 and Click-RT2 have until the next action raises *postfill* and *postdrain*. This gives the Click RT constraints more time to be fulfilled.

Although the extra time afforded to the Click Link is significant, a more critical consideration is that the local state maintenance in the GasP Link is tightly coupled to the self-resetting cycle of the Joint. The modification we introduce in this section decouples the GasP's local management from the Joint's local management so that each individual GasP Link gains the freedom to drive its statewire as long or as short as its physical Link implementation requires.

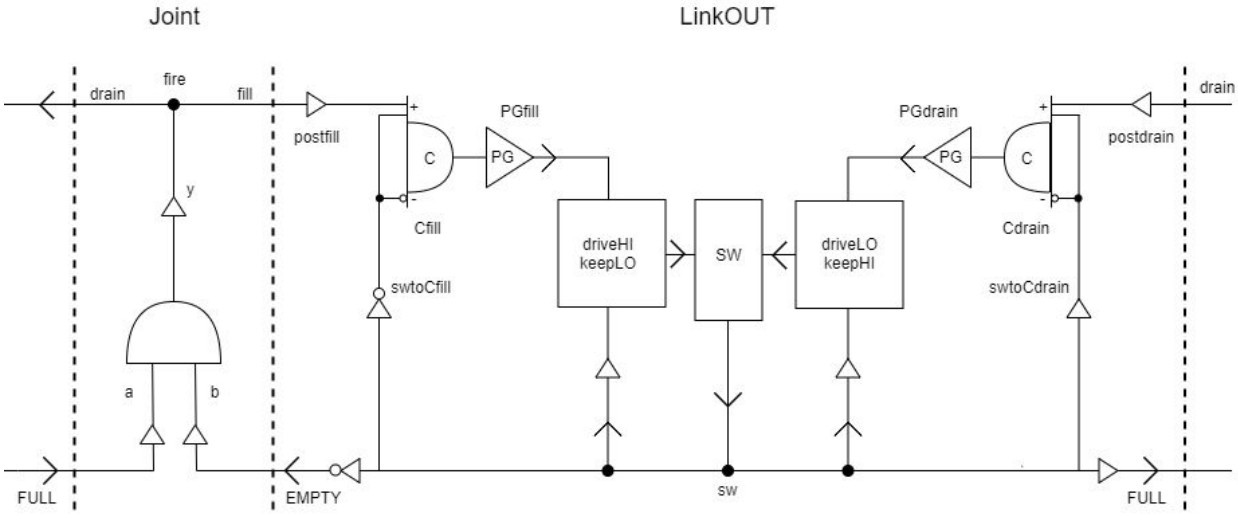

Fig. 7: The LinkOUT and Joint components of the "Click Link - Joint - GasP Link" Model with an edge-triggered GasP (eGasP) Link.

Fig. 7 shows the new Joint-LinkOUT portion of Fig. 5, after replacing the GasP Link with an edge-triggered version called eGasP. The new eGasP Link uses three extra gates at each end: an asymmetric C-element, a Proebsting pulse generator, and a buffer. In Fig. 7, C-element *Cfill* receives its input from buffers *postfill* and *swtoCfill*. The output of *Cfill* drives Proebsting pulse generator *PGfill*. Likewise, C-element *Cdrain* receives its inputs from buffers *postdrain* and *swtoCdrain*, and drives Proebsting pulse generator *PGdrain*. The NuSMV implementation of module eGasP can be found in Appendix B1.

By driving each Proebsting pulse generator with its own asymmetric C-element *Cfill* and *Cdrain*, the internal Link action is sheltered from any external restrictions imposed by the Joint's self-resetting cycle (with the exception of the Joint's RT constraints). Pulse generators drive the eGasP Link for just as long as it needs to perform its state and data actions. The RT Constraints from Section III-C are updated as follows.

eGasP-RT1:

We require that the far-end keepers are activated in time to maintain the appropriate FULL or EMPTY state between fills and drains.

a. If the *fill* signal goes high, buffer *swtokeepHI* must go high before *PGfill* ends its (HI) pulse, i.e., goes low again.

fill+ → swtokeepHI+ < PGfill-

b. If the *drain* signal goes high, buffer *swtokeepLO* must go low before *PGdrain* ends its (HI) pulse, i.e., goes low again.

drain+ → swtokeepLO- < PGdrain-

eGasP-RT2:

We require that the near-end keepers are deactivated to avoid a fight between the near- and far-end keepers.

a. If the *fill* signal goes high, buffer *swtokeepLO* must go high before *PGfill* goes low again.

fill+ → swtokeepLO+ < PGfill-

b. If the *drain* signal goes high, buffer *swtokeepHI* must go low before *PGdrain* goes low again.

drain+ → swtokeepHI- < PGdrain-

As before, we require that there is no conflict over the FULL or EMPTY state of the Link. To achieve this, the output of the Proebsting pulse generator must go low before the output of the opposite pulse generator can go high – as in Link GasP, this is to avoid a state fight. Also as before in Link GasP, the influence of the prior state to the Joint must be disabled by making the corresponding AND input, *a* or *b*, to the Joint low. Finally, and this is a new requirement for Link eGasP, gates *Cfill* and *Cdrain* must be deactivated by making the corresponding postfill and postdrain low before *swtoCfill* and *swtoCdrain* rise again.

eGasP-RT3:

We require that the HI pulses of the two pulse generators do not overlap.

a. Once the *fill* signal goes high, the output of the Proebsting pulse generator *PGfill*, the signal *b* of the Joint, and buffer *postfill* must all go low before the output of the opposite pulse generator *PGdrain* can go high.

fill+ → {PGfill-, Joint.b-, postfill-} < PGdrain+

b. Once the *drain* signal goes high, the output of the Proebsting pulse generator *PGdrain*, the signal *a* of some implied JointOut, and buffer *postdrain* must all go low before the output of the opposite pulse generator *PGfill* can go high.

drain+ → {PGdrain-, impliedJointOut.a-, postdrain-}
< PGfill+

In addition to these updated RT constraints, a new RT constraint is needed to guarantee that the Proebsting's input is used to propagate exactly one cycle of action, preventing a "mismatch" of *fill* and *drain* signals similar to the Joint's FULL and EMPTY signals in Section III-A.

eGasP-RT4:

We require that a Proebsting pulse generator's input is reset before its output is reset to ensure its signal propagates a single cycle of action.

a. If the *fill* signal goes high, the output signal of C-element *Cfill* must go low before *PGfill* can go low again.

fill+ → Cfill- < PGfill-

b. If the *drain* signal goes high, the output signal of C-element *Cdrain* must go low before *PGdrain* can go low again.

drain+ → Cdrain- < PGdrain-

If these updated constraints seem more elaborate than the original RT constraints for the GasP Link, then this is only because this thesis does not yet include the final modification: turning *swtoCfill-Cfill-PGfill* and *swtoCdrain-Cdrain-PGdrain* each into a single gate module with internal RT constraints. In this final modification, the majority of the RT constraints eGasP-RT1 to eGasP-RT4 can be adjusted automatically.

The flip-flops in Click are a partial example of such a modification. The RT constraints related to their edge-triggered behavior are invisible to the designer, as are the internal circuits and assumptions that make the flip-flops edge-triggered [14]. As a result, the designer may be required to satisfy minimum pulse widths on the flip-flop's enable signals, but that is all. Under an assumption that we have this final modification for *swtoCfill-Cfill-PGfill* and *swtoCdrain-Cdrain-PGdrain* in eGasP, the designer sees only their input signals, *postfill* and *postdrain*, and thus must satisfy only the minimum pulse widths on these input signals. Now, these updated constraints with their final (mostly hidden and automated) modifications are indeed simpler than the original RT constraints for the GasP Link. By making GasP Links edge-triggered with modified component eGasP, both Link families now have the time they need to maintain their states. The eGasP Link can be driven as long or short as needed, independent of the self-resetting cycle time in the Joint. The eGasP version increases the Link's modularity and flexibility.

V-B. Synchronizing the Joint reset action with both Links

The adjustment to the Joint was motivated by analyzing its RT constraints in the context of mixed Link circuit families. In particular, Joint-RT1 assumes tightly coupled action reset responses from its Links. This is less of an issue when the LinkIN and LinkOUT are of the same or similar circuit types because, when their activation times coincide, they are also likely to respond at similar times. However, for mixed circuit families, their blueprints may be sufficiently different such that satisfying Joint-RT1 requires extra delay insertion. To avoid this potential required delay insertion, we eliminate the need for RT constraint Joint-RT1 by postponing the Joint reset action until both LinkIn and LinkOUT have responded to the Joint, communicating their new internal states.

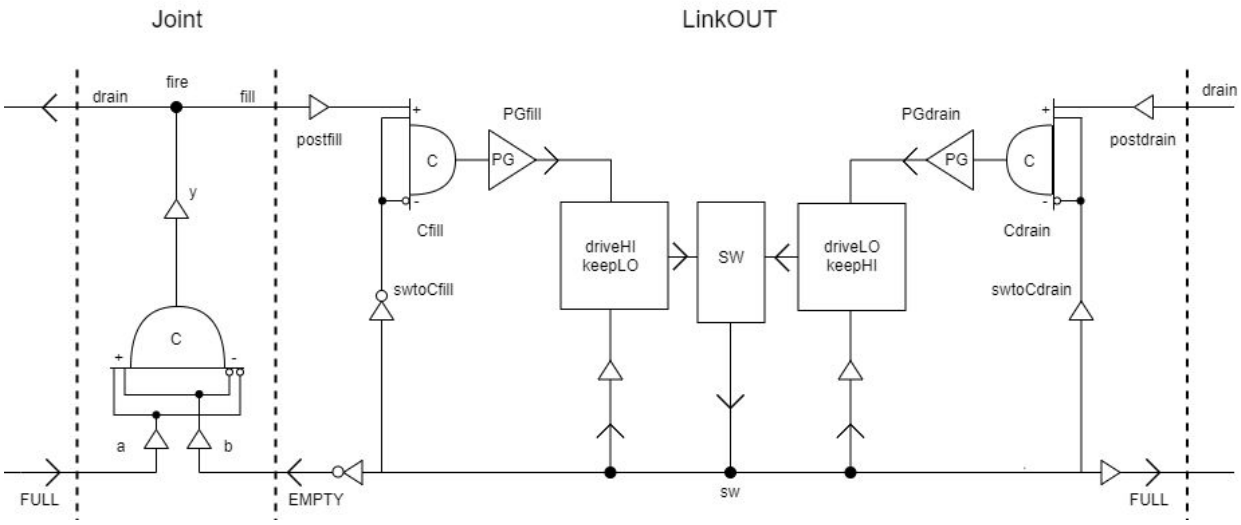

Fig. 8: The LinkOUT and Joint components of the "Click Link - Joint - GasP Link" Model with AND-replacing symmetric C-element in the Joint component and an edge-triggered GasP Link.

Following the eGasP modification, the mixed family FIFO model from Fig. 5 is further updated to reflect the replacement of the AND gate by a symmetric C-element in the Joint component. In its NuSMV implementation, the C-element rises at the conjunction of *a* and *b* (which respectively correspond to signals FULL and EMPTY), and it falls at the conjunction of their inversions !*a* and !*b*. When a and b differ, the output of the C-element remains unchanged. The NuSMV implementation of the Joint with the C-element can be found in Appendix C1.

Now, by design, the Joint resets only after the Links show evidence that they are fulfilling their necessary transportation duties. The RT constraint Joint-RT1 in Section III-A is now implemented by design, i.e., it is vacuously true. The C-element ensures that once signal *fire* has gone high, signals FULL and EMPTY will both go low before signal *fire* can go low again. In addition to guaranteeing Joint-RT1, the C-element allows us to simplify Joint-RT2 to Joint-RT2' by removing a- and b- from the early events.

Joint-RT2':

> We require that once signal *fire* has gone high, signals *postdrain* from LinkIN and *postfill* from LinkOUT go low before both signals *a* and *b* can go high again.
>
> fire+ → {LinkIN.postdrain-, LinkOUT.postfill-} < a+, b+

Because the output of the C-element in the Joint only changes when both its Link inputs have communicated their appropriate new states, it is sufficient to simply constrain that once the Joint has performed its action, it must reset the *postdrain* and *postfill* signals going into LinkIN and LinkOUT respectively before it can begin a new action.

V-C. Next steps for the model modifications

Looking forward beyond the modifications discussed in Section V-A and Section V-B, we further propose implementing both the Click and eGasP Links with four-phase handshake protocols. If we imagine our model to extend as a finite chain of Links and Joints such that each Joint is followed by a Link and each Link is followed by a Joint, we can picture a Link with a near-end Joint and far-end Joint FIFO buffer as in Fig. 9.

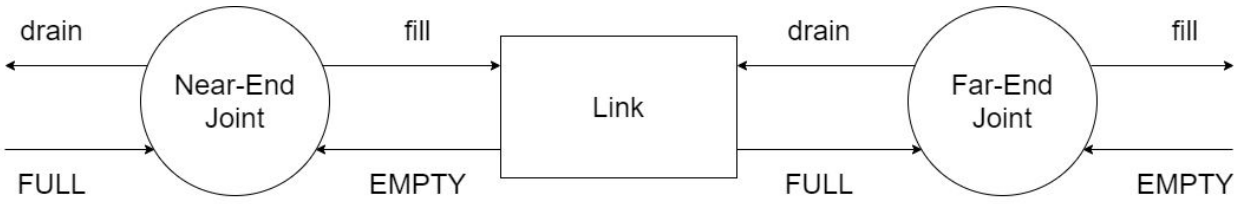

Fig. 9: A First-In-First-Out (FIFO) Buffer, Near-End Joint - Link - Far-End Joint.

A concern with this model is that a far-end Joint may interfere with the near-end Joint action because both Click and GasP Links make the near-end FULL or EMPTY signal low in parallel to making the far-end EMPTY or FULL signal high. Thus, like in Section III-A with the Joint, the signals may become mismatched if a far-end Joint attempts to *drain* the Link before it has been *filled* by the near-end Joint. Similarly, a mismatch may also occur if a near-end Joint attempts to *fill* the Link before it has been *drained* by the far-end Joint. However, by making the Click and eGasP Link four-phase, we can execute the near- and far-end Joint actions in series rather than in parallel. The near-end EMPTY signal is made low first (when *fill* and *drain* signals of the near-end Joint go high), and then the far-end FULL signal can go low (when the *fill* and *drain* signals of the near-end Joint go low, signifying that all Link inputs to the near-end Joint's action have reset.)

These four-phase communication changes will make it easier to satisfy Joint-RT2' because of the extra time available to reset both *postfill* and *postdrain* before the next action becomes enabled. Four-phase handshaking will also facilitate the near-to-far-end relations expressed by RT3 for Click, GasP, and eGasP. Although we theorize that this additional modification will provide further design advantages, implementing the Click and eGasP models as four-phase protocols is outside the scope of this thesis and has been left for future work.

VI. SUMMARY AND CONCLUSION

This thesis presented an abstract circuit model for a simple Link-Joint-Link FIFO buffer with mixed link control circuitry implemented in NuSMV, as well as the relative timing constraints necessary to ensure the correct ordering of computation events for this model. We have analyzed our initial implementations and their respective relative timing constraints to determine the appropriate modifications to help facilitate and simplify the correct mixing of the Click and GasP circuit families. By verifying our new constraints mathematically in NuSMV, we have demonstrated the validity and efficacy of our adjusted models.

Our goal with this project was to ease the design challenge of delay insertion for the Link and Joint Model by incrementally introducing small changes to our FIFO buffer models. To this end, mixing the link control circuitry of Click and GasP is advantageous because these families are so dissimilar. While the GasP circuitry is fast, it can be difficult to work with using standard tools because GasP uses latches and custom-designed logic gates. Click in contrast is slower but much easier to work with because it uses standard logic gate designs. Because Click and GasP lie at the extreme-standard versus extreme-custom ends of the design spectrum of self-timed circuit families, we expect that our work can be ported to other commonly used self-timed circuit families.

With this work, we aim to help facilitate the mix, match, and reuse of existing self-timed solutions between different research teams. Specifically, this thesis enables the reuse of designs independent of the family in which they were implemented. Although our focus for this project was exemplifying reuse and portability for Click and GasP, our position statement is that this work can be broadened and extended to other families.

REFERENCES

[1] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[2] K. Stevens, R. Ginosar, and S. Rotem, "Relative timing," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, 1999, pp. 208–218.

[3] K. S. Stevens, R. Ginosar, and S. Rotem, "Relative timing," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 1, pp. 129–140, Apr. 2003.

[4] H. Park, "Formal Modeling and Verification of Delay-Insensitive Circuits," PhD Thesis, Portland State University, Portland, OR, 2015.

[5] A. Peeters, F. te Beest, M. De Wit, and W. Mallon, "Click elements: An implementation style for data-driven compilation," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, 2010, pp. 3–14.

[6] I. Sutherland and S. Fairbanks, "GasP: A minimal FIFO control," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, 2001, pp. 46–53.

[7] M. Roncken, S. Mettala Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, "Naturalized communication and testing," in *Proceedings - International Symposium on Asynchronous Circuits and Systems*, 2015, pp. 77–84.

[8] M. Roncken *et al.*, "How to think about self-timed systems," in *Conference Record of 51st Asilomar Conference on Signals, Systems and Computers, ACSSC 2017*, 2018, vol. 2017-October, pp. 1597–1604.

[9] M. Roncken and I. Sutherland, "Design and test of high-speed asynchronous circuits," Chapter 7 in *Asynchronous Circuit Applications*, J. Di and S. Smith, Eds. Institution of Engineering & Technology, 2019.

[10] *NuSMV 2.5.4.* (2011). [Online]. Available: http://nusmv.fbk.eu/

[11] H. Park, A. He, M. Roncken, X. Song, and I. Sutherland, "Modular Timing Constraints for Delay-Insensitive Systems," *Journal of Computer Science and Technology*, vol. 31, no. 1, pp. 77–106, Jan. 2016.

[12] R. Cavada et al. *NuSMV 2.5 User Manual*, 2.5 ed. (2010). Accessed: May 20, 2020. [Online]. Available: http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf

[13] H. Park, A. He, M. Roncken, and X. Song, "Semi-modular delay model revisited in context of relative timing," *Electronic Letters*, vol. 51, no. 4, pp. 332–334, Feb. 2015.

[14] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Addison-Wesley Publishing Company, 2011.

Appendix A

NuSMV Implementation and Results of Fig. 5: The Mixed Family FIFO Model

“Click Link - Joint - GasP Link”

This Appendix consists of the following components:

A1: the library containing the NuSMV code for all of the gates and modules in Fig. 5.
pg. 31 - 37

A2: the main program representing the FIFO in Fig. 5 using the library in A1.
pg. 38 - 40

A3: NuSMV results of running our Click Link - Joint - GasP Link model with wall-clock execution time, reachable state space (complexity of design), and showing that the design is correct (i.e. all properties are TRUE).
pg. 41

```
1  ------------------BEGIN LIBRARY
2  --BEGIN NuSMV Library
3  --Allie pre_MixMadeEasy_changes
4  --Honors Thesis May 2020
5
6
7  -- Library modules in here are formally instantiated as follows:
8  --   process cgate        (set, init_val, stop_rise, stop_fall)
9  --   process cgate_lazyHI (set, init_val, stop_rise, stop_fall)
10 --   process FF           (en, set, init_val)
11 --   process DHKL         (inHI, inLO, init_val)
12 --   process DLKH         (inHI, inLO, init_val)
13 --   process SW           (inDHKL, inDLKH, init_val)
14 --   process rt           (eventPOD, eventEARLY, init_rt)
15 --   process Joint        (FULL, EMPTY, stopfall_fire, stoprise_andin)
16 --   process Link_Click   (fill, drain, init_state,
17 --                         stoprise_postfill, stoprise_postdrain,
18 --                         stop_xnorinfar, stop_xorinfar)
19 --   process Link_GasP    (fill, drain, init_val,
20 --                         stoprise_postfill, stoprise_postdrain,
21 --                         stopfall_postfill, stopfall_postdrain)
22
23
24
25 MODULE cgate (set, init_val, stop_rise, stop_fall)
26 --Model for any combinational gate
27 --set:         Boolean function
28 --             specifying what val wants to be in terms of the present inputs
29 --val:         Boolean output
30 --init_val:    initial Boolean value for val
31 --stop_rise:   Boolean function indicating whether or not (a low) val can rise;
32 --             val can rise only if stop_rise is FALSE
33 --stop_fall:   Boolean function indicating whether or not (a high) val fall;
34 --             val can rise only if stop_fall is FALSE
35 --semimodular: Boolean function tracking if changes are done before being disabled
36 --
37 --NOTE:
38 --Difference between using ASSIGN versus VAR for changing module variables:
39 --   ASSIGN is for initialization
40 --          and for private module variables that change only
41 --          when the module is selected for evaluation
42 --          uses :=
43 --   TRANS  is for monitor or random variables
44 --          that must be evaluated each step in the system execution
45 --          uses =
46 --
47    VAR
48       val         : boolean;
49       semimodular : boolean;
50    ASSIGN
51       init(val) := init_val;
52       init(semimodular) := TRUE;
53       next(val) := case
54          --if the val:=set change is unconstrained by a stop, then change
55          (!stop_rise & set & !val) | (!stop_fall & !set & val): set;
56          --otherwise don't change
57          TRUE: val;
58       esac;
59    TRANS
60       next(semimodular) = case
61          --if a val change is disabled by "bullying the change away"
62          --i.e., by only changing inputs
63          --then the semimodular property is ruined
64          ((!stop_rise & set & !val) & next(!(!stop_rise & set) & !val))
65          |
66          ((!stop_fall & !set & val) & next(!(!stop_fall & !set) & val)) : FALSE;
```

```
          TRUE : semimodular;
       esac;
    --PROPERTIES
      --safety
        CTLSPEC AG semimodular
      --progress
        --every module instance is selected for evaluation within finite time
        FAIRNESS running
--END MODULE cgate



MODULE cgate_lazyHI (set, init_val, stop_rise, stop_fall)
--Model for combinational gate where output may never go HI
--set:         Boolean function specifying what the result wants to be
--             in terms of the present inputs
--             When set!=val, changing val takes finite time for !set,
--             but may take infinite time for set.
--val:         Boolean output
--init_val:    initial Boolean value for val
--stop_rise:   Boolean function
--             indicating whether or not (a low) val cannot yet rise
--             val can rise only if stop_rise is FALSE
--stop_fall:   Boolean function
--             indicating whether or not (a high) val cannot yet fall
--             val can rise only if stop_fall is FALSE
--semimodular: Boolean function
--             tracking if changes are done before being disabled
--
   VAR
      val         : boolean;
      semimodular : boolean;
   ASSIGN
      init(val) := init_val;
      init(semimodular) := TRUE;
      next(val) := case
         --if the val:=set change is unconstrained by a stop, then change
         !stop_rise & set & !val : {FALSE, TRUE};
         --but allow a val:=HI change to take forever
         !stop_fall & !set & val : FALSE;
         --otherwise don't change
         TRUE: val;
      esac;
   TRANS
      next(semimodular) = case
         --if a val change is disabled by "bullying the change away"
         --i.e., by only changing inputs
         --then the semimodular property is ruined
         ((!stop_rise & set & !val) & next(!(!stop_rise & set) & !val))
         |
         ((!stop_fall & !set & val) & next(!(!stop_fall & !set) & val)) : FALSE;
         TRUE : semimodular;
      esac;
   --PROPERTIES
     --safety
       CTLSPEC AG semimodular
       --if set!=val and set holds,
       --then there is a future where changing val takes finite time
       --and there is a future where changing val takes infinite time,
       --i.e., where !val holds forever
       CTLSPEC AG ((!stop_rise & set & !val) -> (EX val))
       CTLSPEC AG ((!stop_rise & set & !val) -> (EX !val))
     --progress
       --every module instance is selected for evaluation within finite time
       FAIRNESS running
--END MODULE cgate_lazyHI
```

```
MODULE FF (en, set, init_val)
--Model for edge-triggered flipflop
--en       : Boolean function whose rising edge enables the flipflop
--set      : Boolean function specifying what the result wants to be
--            in terms of the present inputs
--Q        : Boolean output
--init_val : initial Boolean value for Q
--
   VAR
      Q : boolean;
   ASSIGN
      init(Q) := init_val;
   TRANS
      next(Q) = case
         !en & next(en): set;
         TRUE: Q;
      esac;
   --PROPERTIES
     --safety
       --Semimodularity is achieved by design (TRANS)
     --progress
       --Fairness running is automatic
       --because Q is evaluated each step by design (TRANS)
--END MODULE FF



MODULE DHKL (inHI, inLO, init_val)
--Model for GasP pull-oneway-keep-theotherway gate
--inHI      : Boolean function for driving val HI
--inLO      : Boolean function for keeping val LO
--val       : {driveHI, keepLO, tristate}
--init_val  : Boolean function indicating an initial value for val
--
   VAR
      val : {driveHI, keepLO, tristate};
   ASSIGN
      init(val) := case
         init_val : tristate; --some other module in the design keeps HI
         TRUE     : keepLO;
      esac;
   TRANS
      next(val) = case
         --the ordering assumes that inHI and inLO
         --are never TRUE at the same time
         --(this will be checked in PROPERTIES)
         inHI : driveHI;
         inLO : keepLO;
         TRUE: tristate;
      esac;
   --PROPERTIES
     --safety
       --Semimodularity is achieved by design (TRANS)
       --Check that inHI and inLO are never TRUE at the same time
         CTLSPEC AG !(inHI & inLO)
     --progress
       --Fairness running is automatic
       --because val is evaluated each step by design (TRANS)
--END MODULE DHKL



MODULE DLKH (inHI, inLO, init_val)
```

```
--Model for GasP pull-oneway-keep-theotherway gate
--inHI      : Boolean function for keeping val HI
--inLO      : Boolean function for driving val LO
--val       : {driveLO, keepHI, tristate}
--init_val  : Boolean function indicating an initial value for val
--
   VAR
      val : {driveLO, keepHI, tristate};
   ASSIGN
      init(val) := case
         init_val : keepHI;
         TRUE     : tristate; --some other module in the design keeps LO
      esac;
   TRANS
      next(val) = case
         --the ordering assumes that inHI and inLO
         --are never TRUE at the same time
         --(will be checked in PROPERTIES)
         inHI : keepHI;
         inLO : driveLO;
         TRUE : tristate;
      esac;
   --PROPERTIES
     --safety
       --Semimodularity is achieved by design (TRANS)
       --Check that inHI and inLO are never TRUE at the same time
         CTLSPEC AG !(inHI & inLO)
     --progress
       --Fairness running is automatic
       --because val is evaluated each step by design (TRANS)
--END MODULE DLKH



MODULE SW (inDHKL, inDLKH, init_val)
--Model for GasP statewire
--inDHKL     : {driveHI, keepLO, tristate}
--inDLKH     : {driveLO, keepHI, tristate}
--val        : Boolean output
--init_val   : initial Boolean value for val
--noFight    : Boolean tracking that the inputs never conflict
--noFloat    : Boolean tracking that the inputs never both float
--
   VAR
      val     : boolean;
      noFight : boolean;
      -- monitors if inDHKL and inDLKH never have conflicting HI-LO indicators
      noFloat : boolean;
      -- monitors if inDHKL and inDLKH never both float at the same time
   DEFINE
      goHI := (inDHKL=tristate & inDLKH=keepHI)
              | (inDHKL=driveHI & inDLKH=tristate)
              | (inDHKL=driveHI & inDLKH=keepHI);
      goLO := (inDHKL=tristate & inDLKH=driveLO)
              | (inDHKL=keepLO  & inDLKH=tristate)
              | (inDHKL=keepLO  & inDLKH=driveLO);
   ASSIGN
      init(val)  := init_val;
      init(noFight) := TRUE;
      init(noFloat) := TRUE;
   TRANS
      next(val) = case
         goHI: TRUE;
         goLO: FALSE;
         TRUE: val;
         --val value doesn't matter for TRUE,
```

```
265            --because we never want to get here (fight or float)
266         esac;
267      TRANS
268         next(noFight) = case
269            (inDHKL=driveHI & inDLKH=driveLO)
270            | (inDHKL=keepLO & inDLKH=keepHI): FALSE;
271            TRUE: noFight;
272         esac;
273      TRANS
274         next(noFloat) = case
275            inDHKL=tristate & inDLKH=tristate : FALSE;
276            TRUE: noFloat;
277         esac;
278      --PROPERTIES
279        --safety
280          CTLSPEC AG noFight;
281          CTLSPEC AG noFloat;
282       --progress
283         --Fairness running is automatic
284         --because val is evaluated each step by design (TRANS)
285   --END MODULE SW
286
287
288
289   MODULE rt (eventPOD, eventEARLY, init_rt)
290   --eventPOD  : Boolean function
291   --            specifying the point of divergence for this constraint
292   --eventEARLY: Boolean function
293   --            specifying the early event, releasing the constraint
294   --init_rt   : {GREEN, RED} initialization state for stoplight
295   --stoplight : {GREEN, RED} rt state
296   --stop      : Boolean function that can be used to prevent the late event
297   --            via stop_rise or stop_fall
298   --            in library gates with these formal parameters
299   --NOTE:
300   --Simplifying assumption for Allie's circuits
301   --so we don't have to use the more complex rt from Hoon's thesis:
302   --ASSUMPTION:
303   -- Per rt, the three POD/early/late functions
304   -- differ with at least one state distance
305   --
306      VAR
307         stoplight : {GREEN, RED};
308      ASSIGN
309         init(stoplight) := init_rt;
310      TRANS
311         next(stoplight) = case
312            myEARLY                 : GREEN;
313            stoplight=GREEN & myPOD : RED;
314            TRUE                    : stoplight;
315         esac;
316      DEFINE
317         myPOD   := !eventPOD & next(eventPOD);
318                    --looks for a low to high level change
319         myEARLY := !eventEARLY & next(eventEARLY);
320                    --looks for a low to high level change
321         stop    := (stoplight=RED);
322      --PROPERTIES
323        --safety
324          --Semimodularity is achieved by design (TRANS)
325        --progress
326          --Fairness running is automatic
327          --because val is evaluated each step by design (TRANS)
328   --END MODULE rt
329
330
```

```
MODULE Joint (FULL, EMPTY, stopfall_fire, stoprise_andin)
--FULL     : Boolean from input Link
--EMPTY    : Boolean from output Link
--fill     : Boolean to output Link
--drain    : Boolean to input Link
--stop*    : Boolean imported RT constraint stops
--NOTE:
--   assume that everything is reset initially,
--   no new communications have come in yet
   VAR
      buf_postfull   : process cgate (FULL,  FALSE, stoprise_andin, FALSE);
      buf_postempty  : process cgate (EMPTY, FALSE, stoprise_andin, FALSE);
      and            : process cgate (andin1 & andin2, FALSE, FALSE, FALSE);
      buf_postand    : process cgate (and.val, FALSE, FALSE, stopfall_fire);
   DEFINE
      andin1:= buf_postfull.val;
      andin2:= buf_postempty.val;
      fire  := buf_postand.val;
      fill  := fire;
      drain := fire;
   --PROPERTIES
     --safety
       --semimodularity is checked within each cgate
     --progress
       --every module instance is selected for evaluation within finite time
       FAIRNESS running
--END MODULE Joint



MODULE Link_Click (fill, drain, init_state,
                   stoprise_postfill, stoprise_postdrain,
                   stop_xnorinfar, stop_xorinfar)
--fill      : Boolean input from sending Joint
--drain     : Boolean input from receiving Joint
--init_state: TRUE for FULL with req=1 and ack=0,
--            FALSE for EMPTY with req=0 and ack=0
--EMPTY     : Boolean output to sending Joint
--FULL      : Boolean output to receiving Joint
--stop*     : Boolean imported RT constraint stops
   VAR
      buf_postfill       : process cgate
                           (fill,  FALSE, stoprise_postfill, FALSE);
      buf_postdrain      : process cgate
                           (drain, FALSE, stoprise_postdrain, FALSE);
      xnor_gate          : process cgate
                           (buf_FFreqtoreqnear.val xnor buf_FFacktoackfar.val,
                            !init_state, FALSE, FALSE);
      xor_gate           : process cgate
                           (buf_FFreqtoreqfar.val xor buf_FFacktoacknear.val,
                            init_state, FALSE, FALSE);
      --
      FFreq              : process FF
                           (buf_postfill.val, !buf_FFreqtoFFreq.val, init_state);
      buf_FFreqtoFFreq   : process cgate
                           (FFreq.Q, init_state, FALSE, FALSE);
      buf_FFreqtoreqnear : process cgate
                           (FFreq.Q, init_state, FALSE, FALSE);
      buf_FFreqtoreqfar  : process cgate
                           (FFreq.Q, init_state, stop_xorinfar, stop_xorinfar);
      --
      FFack              : process FF
                           (buf_postdrain.val, !buf_FFacktoFFack.val, FALSE);
      buf_FFacktoFFack   : process cgate
                           (FFack.Q, FALSE, FALSE, FALSE);
```

```
      buf_FFacktoacknear : process cgate
                           (FFack.Q, FALSE, FALSE, FALSE);
      buf_FFacktoackfar  : process cgate
                           (FFack.Q, FALSE, stop_xnorinfar, stop_xnorinfar);
   DEFINE
      EMPTY    := xnor_gate.val;
      FULL     := xor_gate.val;
      postfill := buf_postfill.val;
      postdrain:= buf_postdrain.val;
   --PROPERTIES
     --progress
       --every module instance is selected for evaluation within finite time
       FAIRNESS running
--END MODULE Link_Click



MODULE Link_GasP (fill, drain, init_val,
                  stoprise_postfill, stoprise_postdrain,
                  stopfall_postfill, stopfall_postdrain)
--fill     : Boolean input from sending Joint
--drain    : Boolean input from receiving Joint
--init_val : TRUE for FULL, FALSE for EMPTY
--EMPTY    : Boolean output to sending Joint
--FULL     : Boolean output to receiving Joint
--stop*    : Boolean imported RT constraint stops
   VAR
      buf_postfill      : process cgate
                          (fill, FALSE, stoprise_postfill, stopfall_postfill);
      buf_postdrain     : process cgate
                          (drain, FALSE, stoprise_postdrain, stopfall_postdrain);
      --
      driveHIkeepLO     : process DHKL
                          (postfill, !postfill & !swtokeepLO, init_val);
      driveLOkeepHI     : process DLKH
                          (!postdrain & swtokeepHI, postdrain, init_val);
      statewire         : process SW
                          (driveHIkeepLO.val, driveLOkeepHI.val, init_val);
      --
      buf_swtokeepLO    : process cgate (sw,  init_val, FALSE, FALSE);
      buf_swtokeepHI    : process cgate (sw,  init_val, FALSE, FALSE);
      buf_swtoEMPTY     : process cgate (!sw,!init_val, FALSE, FALSE);
      buf_swtoFULL      : process cgate (sw,  init_val, FALSE, FALSE);
   DEFINE
      sw          := statewire.val;
      EMPTY       := buf_swtoEMPTY.val;
      FULL        := buf_swtoFULL.val;
      swtokeepHI  := buf_swtokeepHI.val;
      swtokeepLO  := buf_swtokeepLO.val;
      postfill    := buf_postfill.val;
      postdrain   := buf_postdrain.val;
   --PROPERTIES
     --progress
       --every module instance isselected for evaluation within finite time
       FAIRNESS running
--END MODULE Link_GasP
------------------END LIBRARY
```

```
----------------BEGIN MAIN PROGRAM Click-Joint-GasP
MODULE main
   VAR
      thisJoint : process Joint
                  (FULL, EMPTY, Joint_stopfall_fire, Joint_stoprise_andin);
                  --starts reset

      LinkIn    : process Link_Click
                  (ENV_fill, drain, TRUE,
                    LinkIn_stoprise_postfill, LinkIn_stoprise_postdrain,
                    LinkIn_stop_xnorinfar, LinkIn_stop_xorinfar);
                   --starts FULL

      LinkOut   : process Link_GasP
                  (fill, ENV_drain, FALSE,
                   LinkOut_stoprise_postfill, LinkOut_stoprise_postdrain,
                   LinkOut_stopfall_postfill, LinkOut_stopfall_postdrain);
                   --starts EMPTY

      ENVIn     : process cgate_lazyHI
                  (ENV_EMPTY, FALSE, FALSE, FALSE);
                   --starts reset

      ENVOut    : process cgate_lazyHI
                  (ENV_FULL, FALSE, FALSE, FALSE);
                   --starts reset
   DEFINE
      fire      := thisJoint.fire;
      fill      := thisJoint.fill;
      drain     := thisJoint.drain;
      FULL      := LinkIn.FULL;
      EMPTY     := LinkOut.EMPTY;
      ENV_FULL  := LinkOut.FULL;
      ENV_EMPTY := LinkIn.EMPTY;
      ENV_fill  := ENVIn.val;
      ENV_drain := ENVOut.val;

   --PROPERTIES
      --safety
        --Semimodularity  and other properties are checked within each process
      --progress
        --every module instance is selected for evaluation within finite time
        FAIRNESS running
        --there's real action!
        CTLSPEC AG (fill   -> (EF !fill));
        CTLSPEC AG (!fill  -> (EF fill));
        CTLSPEC AG (drain  -> (EF !drain));
        CTLSPEC AG (!drain -> (EF drain));



   --RT constaints for thisJoint:
   --
   --(1) Joint-RT1:
   --    fire+ -> !EMPTY, !FULL < fire-
   VAR
      Joint-RT1a : process rt (fire, !EMPTY, GREEN);
      Joint-RT1b : process rt (fire, !FULL , GREEN);
   DEFINE
      Joint_stopfall_fire := Joint-RT1a.stop | Joint-RT1b.stop;
   --
   --(2) Joint-RT2:
   --    fire+
   --    -> thisJoint.andin1-, thisJoint.andin2-,
   --       LinkIn.postdrain-, LinkOut.postfill-
   --    <  thisJoint.andin1+, thisJoint.andin2+
```

```
67      VAR
68         Joint-RT2a: process rt (fire, !thisJoint.andin1, GREEN);
69         Joint-RT2b: process rt (fire, !thisJoint.andin2, GREEN);
70         Joint-RT2c: process rt (fire, !LinkIn.postdrain, GREEN);
71         Joint-RT2d: process rt (fire, !LinkOut.postfill, GREEN);
72      DEFINE
73         Joint_stoprise_andin := Joint-RT2a.stop | Joint-RT2b.stop |
74                                 Joint-RT2c.stop | Joint-RT2d.stop;
75
76
77
78      --RT constraints for LinkIn (Click):
79      --
80      --(1) Click-RT1:
81      --    LinkIn.FFreq.Q+ -> LinkIn.buf_FFreqtoFFreq.val+ < LinkIn.postfill+
82      --    LinkIn.FFreq.Q- -> LinkIn.buf_FFreqtoFFreq.val- < LinkIn.postfill+
83      VAR
84        Click-RT1a : process rt (LinkIn.FFreq.Q,  LinkIn.buf_FFreqtoFFreq.val,  GREEN);
85        Click-RT1b : process rt (!LinkIn.FFreq.Q, !LinkIn.buf_FFreqtoFFreq.val, GREEN);
86      DEFINE
87        LinkIn_stoprise_postfill := Click-RT1a.stop | Click-RT1b.stop;
88      --
89      --(2) Click-RT2:
90      --    LinkIn.FFack.Q+ -> LinkIn.buf_FFacktoFFack.val+ < LinkIn.postdrain+
91      --    LinkIn.FFack.Q- -> LinkIn.buf_FFacktoFFack.val- < LinkIn.postdrain+
92      VAR
93        Click-RT2a : process rt (LinkIn.FFack.Q,  LinkIn.buf_FFacktoFFack.val,  GREEN);
94        Click-RT2b : process rt (!LinkIn.FFack.Q, !LinkIn.buf_FFacktoFFack.val, GREEN);
95      DEFINE
96        LinkIn_stoprise_postdrain := Click-RT2a.stop | Click-RT2b.stop;
97      --
98      --(3) Click-RT3:
99      --    ENV_fill+ (i.e., LinkIn.fill+)
100     --    -> LinkIn.ImpliedJointIn.andin2-
101     --       i.e., here propagation stops at LinkIn.postfill-
102     --    <  LinkIn.buf_FFacktoackfar+
103     --
104     --    drain+ -> thisJoint.andin1- < FFreqtoreqfar+
105     VAR
106        Click-RT3a: process rt (ENV_fill, !LinkIn.postfill, GREEN);
107        Click-RT3b: process rt (drain, !thisJoint.andin1, GREEN);
108     DEFINE
109        LinkIn_stop_xnorinfar := Click-RT3a.stop;
110        LinkIn_stop_xorinfar  := Click-RT3b.stop;
111
112
113
114     --RT constraints for LinkOut (GasP)
115     --
116     --(1) GasP-RT1:
117     --    fill+  -> LinkOut.swtokeepHI+ < LinkOut.postfill-
118     --    ENV_drain+ -> LinkOut.swtokeepLO- < LinkOut.postdrain-
119     VAR
120        GasP-RT1a: process rt (fill,  LinkOut.swtokeepHI,  GREEN);
121        GasP-RT1b: process rt (ENV_drain, !LinkOut.swtokeepLO, GREEN);
122     --
123     --(2) GasP-RT2:
124     --    fill+  -> LinkOut.swtokeepLO+ < LinkOut.postfill-
125     --    ENV_drain+ -> LinkOut.swtokeepHI- < LinkOut.postdrain-
126     VAR
127        GasP-RT2a: process rt (fill,  LinkOut.swtokeepLO,  GREEN);
128        GasP-RT2b: process rt (ENV_drain, !LinkOut.swtokeepHI, GREEN);
129     --
130     DEFINE
131        LinkOut_stopfall_postfill  := GasP-RT1a.stop | GasP-RT2a.stop;
132        LinkOut_stopfall_postdrain := GasP-RT1b.stop | GasP-RT2b.stop;
```

```
133      --
134      -- (3) GasP-RT3:
135      --    fill+  -> LinkOut.postfill-, thisJoint.andin2- < LinkOut.postdrain+
136      --
137      --    ENV_drain+
138      --    -> LinkOut.postdrain-,
139      --       ImpliedJointOut.andin1- (i.e., here again LinkOut.postdrain-)
140      --    <  LinkOut.postfill+
141      VAR
142         GasP-RT3a1: process rt (fill, !LinkOut.postfill,  GREEN);
143         GasP-RT3a2: process rt (fill, !thisJoint.andin2, GREEN);
144         --
145         GasP-RT3b1: process rt (ENV_drain, !LinkOut.postdrain, GREEN);
146      DEFINE
147         LinkOut_stoprise_postfill  := GasP-RT3b1.stop;
148         LinkOut_stoprise_postdrain := GasP-RT3a1.stop | GasP-RT3a2.stop;
149   -- END MODULE main
150   ----------------END MAIN PROGRAM Click-Joint-GasP
151
```

```
Wall-clock execution time: 6 minutes
############################################################################
system diameter: 107
reachable states: 114984 (2^16.8111) out of 2.65633e+021 (2^71.1699)
############################################################################



*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 14:15:02 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
*** Copyright (c) 2010, Fondazione Bruno Kessler
*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson
WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG semimodular IN thisJoint.buf_postfull is true
-- specification AG semimodular IN thisJoint.buf_postempty is true
-- specification AG semimodular IN thisJoint.and is true
-- specification AG semimodular IN thisJoint.buf_postand is true
-- specification AG semimodular IN LinkIn.buf_postfill is true
-- specification AG semimodular IN LinkIn.buf_postdrain is true
-- specification AG semimodular IN LinkIn.xnor_gate is true
-- specification AG semimodular IN LinkIn.xor_gate is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoFFreq is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqnear is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqfar is true
-- specification AG semimodular IN LinkIn.buf_FFacktoFFack is true
-- specification AG semimodular IN LinkIn.buf_FFacktoacknear is true
-- specification AG semimodular IN LinkIn.buf_FFacktoackfar is true
-- specification AG semimodular IN LinkOut.buf_postfill is true
-- specification AG semimodular IN LinkOut.buf_postdrain is true
-- specification AG !(inHI & inLO) IN LinkOut.driveHIkeepLO is true
-- specification AG !(inHI & inLO) IN LinkOut.driveLOkeepHI is true
-- specification AG noFight IN LinkOut.statewire is true
-- specification AG noFloat IN LinkOut.statewire is true
-- specification AG semimodular IN LinkOut.buf_swtokeepLO is true
-- specification AG semimodular IN LinkOut.buf_swtokeepHI is true
-- specification AG semimodular IN LinkOut.buf_swtoEMPTY is true
-- specification AG semimodular IN LinkOut.buf_swtoFULL is true
-- specification AG semimodular IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVIn is true
-- specification AG semimodular IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVOut is true
-- specification AG (fill -> EF !fill)  is true
-- specification AG (!fill -> EF fill)  is true
-- specification AG (drain -> EF !drain)  is true
-- specification AG (!drain -> EF drain)  is true

```

Appendix B

NuSMV Implementation and Results of Full FIFO of Fig. 7: The Mixed Family FIFO Model

"Click Link - Joint - eGasP Link"

This Appendix consists of the following components:

```
1  ------------------BEGIN LIBRARY EXTENSION eGasP
2  --BEGIN NuSMV Library
3  --Allie eGasPextension
4  --Honors Thesis May 2020
5
6
7  -- Library modules in here are formally instantiated as follows:
8  --   C                    (set, reset, init_val, stop_rise, stop_fall)
9  --   ProebstingPulseGen   (set, stop_rise, stop_fall)
10 --   Link_eGasP           (fill, drain, init_val,
11 --                         stoprise_PGfill, stoprise_PGdrain,
12 --                         stopfall_PGfill, stopfall_PGdrain)
13
14
15
16 MODULE C (set, reset, init_val, stop_rise, stop_fall)
17 --Model for any symmetric or asymmetric sequential gate.
18 --
19 --NOTE:
20 --When set = !reset, we have a combinational gate, otherwise a sequential gate
21 --So, in principle, we can specify any cgate in the library using C.
22 --We assume that set and reset are mutually exclusive, and will check for this.
23 --
24 --set:         Boolean function specifying when val can go high
25 --reset:       Boolean function specifying when val can go low
26 --val:         Boolean output
27 --init_val:    initial Boolean value for val
28 --stop_rise:   Boolean function indicating
29 --             whether or not (a low) val can rise;
30 --             val can rise only if stop_rise is FALSE
31 --stop_fall:   Boolean function indicating
32 --             whether or not (a high) val can fall;
33 --             val can rise only if stop_fall is FALSE
34 --semimodular: Boolean function tracking
35 --             if changes are done before being disabled
36 --NOTE:
37 --Difference between using ASSIGN versus VAR for changing module variables:
38 --   ASSIGN is for initialization
39 --          and for private module variables that change only
40 --          when the module is selected for evaluation
41 --          uses :=
42 --   TRANS  is for monitor or random variables
43 --          that must be evaluated each step in the system execution
44 --          uses =
45 --
46    VAR
47       val         : boolean;
48       semimodular : boolean;
49    ASSIGN
50       init(val) := init_val;
51       init(semimodular) := TRUE;
52       next(val) := case
53          --drive HI
54          !stop_rise & set & !val : TRUE;
55          --or LO
56          !stop_fall & reset & val  : FALSE;
57          --or maintain current val state
58          TRUE: val;
59       esac;
60    TRANS
61       next(semimodular) = case
62          --if a val change is disabled by "bullying the change away"
63          --i.e., by only changing inputs
64          --then the semimodular property is ruined
65          ((!stop_rise & set & !val) & next(!(!stop_rise & set) & !val))
66          |
```

```
          ((!stop_fall & reset & val) & next(!(!stop_fall & reset) & val)) : FALSE;
          TRUE : semimodular;
       esac;
    --PROPERTIES
      --safety
       CTLSPEC AG semimodular
       --check that drive_HI and drive_LO are mutually exclusive
       CTLSPEC AG !(set & reset)
      --progress
       --every module instance is selected for evaluation within finite time
       FAIRNESS running
--END MODULE C



MODULE Proebsting (set, stop_rise, stop_fall)
--Proebsting pulsegenerator model
--for making level-triggered Links behave edge-triggered,
--with a to be determined high pulsewidth,
--be it (much) shorter or (much) longer
--than filling or draining Joint cycles.
--
--set:         Boolean enable function
--            HI makes val HI and then LO within finite time
--val:         Boolean output
--stop_rise:   Boolean function indicating
--            whether or not (a low) val may rise;
--            val can rise only if stop_rise is FALSE
--stop_fall:   Boolean function indicating
--            whether or not (a high) val may fall
--semimodular: Boolean function tracking
--            if state changes were done before being disabled
--NOTE:
--Difference between using ASSIGN versus VAR for changing module variables:
--   ASSIGN is for initialization
--          and for private module variables that change only
--          when the module is selected for evaluation
--          uses :=
--   TRANS  is for monitor or random variables
--          that must be evaluated each step in the system execution
--          uses =
--
   VAR
      val         : boolean;
      semimodular : boolean;
   ASSIGN
      init(val)         := FALSE;
      init(semimodular) := TRUE;
      next(val) := case
         --val FALSE to TRUE depends on set
         !stop_rise & set & !val : TRUE;
         --val TRUE to FALSE is independent of set
         !stop_fall & val : FALSE;
         TRUE: val;
      esac;
   TRANS
      next(semimodular) = case
         --if a state change is disabled by "bullying the change away"
         --i.e., by changing inputs only
         --then semimodularity is ruined
         (!stop_rise & set & !val) & next(!(!stop_rise & set) & !val) : FALSE;
         (!stop_fall & val) & next(stop_fall & val) : FALSE;
         TRUE : semimodular;
      esac;
   --PROPERTIES
     --safety
```

```
        CTLSPEC AG semimodular;
        --We would like the pulsegenerator to behave like a buffer
        --so we check that set will become FALSE before val becomes FALSE again
        --To satisfy, this will require relative timing constraints
        CTLSPEC AG ((set & val) -> (A [val U !set]));
      --progress
        --every module instance must be selected for evaluation within finite time
        FAIRNESS running
--END MODULE Proebsting



MODULE Link_eGasP (fill, drain, init_val,
                   stoprise_PGfill, stoprise_PGdrain,
                   stopfall_PGfill, stopfall_PGdrain)
--NOTE:
--Link_eGasP provides edge-triggered versions of Link_GasP
--by using a Proebsting amplifier driven by an asymmetric C element
--tied to fill respectively drain at each end.
--This solution can drive the link (much) shorter or (much) longer
--than the Joint self-resetting cycle at each end permits,
--especially when the Joints have different cycle times.
--We use it to drive the Link just as long as it needs
--to get its state and data across.
--
--fill     : Boolean input from sending Joint
--drain    : Boolean input from receiving Joint
--init_val : TRUE for FULL, FALSE for EMPTY
--EMPTY    : Boolean output to sending Joint
--FULL     : Boolean output to receiving Joint
--stop*    : Boolean imported RT constraint stops
   VAR
      buf_postfill   : process cgate (fill, FALSE, FALSE, FALSE);
      C_fill         : process C
                       (buf_postfill.val & buf_swtoCfill.val, !buf_swtoCfill.val,
                        FALSE, FALSE, FALSE);
      PG_fill        : process Proebsting
                       (C_fill.val, stoprise_PGfill, stopfall_PGfill);
      --
      buf_postdrain  : process cgate (drain, FALSE, FALSE, FALSE);
      C_drain        : process C
                       (buf_postdrain.val & buf_swtoCdrain.val, !buf_swtoCdrain.val,
                        FALSE, FALSE, FALSE);
      PG_drain       : process Proebsting
                       (C_drain.val, stoprise_PGdrain, stopfall_PGdrain);
      --
      driveHIkeepLO  : process DHKL (PG_fill.val, !PG_fill.val & !swtokeepLO, init_val);
      driveLOkeepHI  : process DLKH (!PG_drain.val & swtokeepHI, PG_drain.val, init_val);
      statewire      : process SW   (driveHIkeepLO.val, driveLOkeepHI.val, init_val);
      --
      buf_swtokeepLO : process cgate (sw,  init_val, FALSE, FALSE);
      buf_swtokeepHI : process cgate (sw,  init_val, FALSE, FALSE);
      buf_swtoEMPTY  : process cgate (!sw,!init_val, FALSE, FALSE);
      buf_swtoCfill  : process cgate (!sw,!init_val, FALSE, FALSE);
      buf_swtoFULL   : process cgate (sw,  init_val, FALSE, FALSE);
      buf_swtoCdrain : process cgate (sw,  init_val, FALSE, FALSE);

   DEFINE
      sw          := statewire.val;
      EMPTY       := buf_swtoEMPTY.val;
      FULL        := buf_swtoFULL.val;
      swtokeepHI  := buf_swtokeepHI.val;
      swtokeepLO  := buf_swtokeepLO.val;
      postfill    := buf_postfill.val;
      postdrain   := buf_postdrain.val;
      Cfill       := C_fill.val;
```

```
199         Cdrain      := C_drain.val;
200         PGfill      := PG_fill.val;
201         PGdrain     := PG_drain.val;
202       --PROPERTIES
203         --progress
204           --every module instance is selected for evaluation within finite time
205           FAIRNESS running
206   --END MODULE Link_eGasP
207   -------------------END LIBRARY EXTENSION eGasP
208
209
```

```
----------------BEGIN MAIN PROGRAM Click-Joint-eGasP
MODULE main
   VAR
      thisJoint : process Joint
                  (FULL, EMPTY, Joint_stopfall_fire, Joint_stoprise_andin);
                  --starts reset

      LinkIn    : process Link_Click
                  (ENV_fill, drain, TRUE,
                    LinkIn_stoprise_postfill, LinkIn_stoprise_postdrain,
                    LinkIn_stop_xnorinfar, LinkIn_stop_xorinfar);
                   --starts FULL

      LinkOut   : process Link_eGasP
                  (fill, ENV_drain, FALSE,
                   LinkOut_stoprise_PGfill, LinkOut_stoprise_PGdrain,
                   LinkOut_stopfall_PGfill, LinkOut_stopfall_PGdrain);
                   --starts EMPTY

      ENVIn     : process cgate_lazyHI
                  (ENV_EMPTY, FALSE, FALSE, FALSE);
                   --starts reset

      ENVOut    : process cgate_lazyHI
                  (ENV_FULL, FALSE, FALSE, FALSE);
                   --starts reset
   DEFINE
      fire      := thisJoint.fire;
      fill      := thisJoint.fill;
      drain     := thisJoint.drain;
      FULL      := LinkIn.FULL;
      EMPTY     := LinkOut.EMPTY;
      ENV_FULL  := LinkOut.FULL;
      ENV_EMPTY := LinkIn.EMPTY;
      ENV_fill  := ENVIn.val;
      ENV_drain := ENVOut.val;

   --PROPERTIES
      --safety
        --Semimodularity  and other properties are checked within each process
      --progress
        --every module instance is selected for evaluation within finite time
        FAIRNESS running
        --there's real action!
        CTLSPEC AG (fill   -> (EF !fill));
        CTLSPEC AG (!fill  -> (EF fill));
        CTLSPEC AG (drain  -> (EF !drain));
        CTLSPEC AG (!drain -> (EF drain));



   --RT constaints for thisJoint:
   --
   --(1) Joint-RT1:
   --    fire+ -> !EMPTY, !FULL < fire-
   VAR
      Joint-RT1a : process rt (fire, !EMPTY, GREEN);
      Joint-RT1b : process rt (fire, !FULL , GREEN);
   DEFINE
      Joint_stopfall_fire := Joint-RT1a.stop | Joint-RT1b.stop;
   --
   --(2) Joint-RT2:
   --    fire+
   --    -> thisJoint.andin1-, thisJoint.andin2-,
   --       LinkIn.postdrain-, LinkOut.postfill-
   --    <  thisJoint.andin1+, thisJoint.andin2+
```

```
   VAR
      Joint-RT2a: process rt (fire, !thisJoint.andin1, GREEN);
      Joint-RT2b: process rt (fire, !thisJoint.andin2, GREEN);
      Joint-RT2c: process rt (fire, !LinkIn.postdrain, GREEN);
      Joint-RT2d: process rt (fire, !LinkOut.postfill, GREEN);
   DEFINE
      Joint_stoprise_andin := Joint-RT2a.stop | Joint-RT2b.stop |
                          Joint-RT2c.stop | Joint-RT2d.stop;



--RT constraints for LinkIn (Click):
   --
   --
   --(1) Click-RT1:
   --    LinkIn.FFreq.Q+ -> LinkIn.buf_FFreqtoFFreq.val+ < LinkIn.postfill+
   --    LinkIn.FFreq.Q- -> LinkIn.buf_FFreqtoFFreq.val- < LinkIn.postfill+
   VAR
     Click-RT1a : process rt (LinkIn.FFreq.Q,  LinkIn.buf_FFreqtoFFreq.val,  GREEN);
     Click-RT1b : process rt (!LinkIn.FFreq.Q, !LinkIn.buf_FFreqtoFFreq.val, GREEN);
   DEFINE
     LinkIn_stoprise_postfill := Click-RT1a.stop | Click-RT1b.stop;
   --
   --(2) Click-RT2:
   --    LinkIn.FFack.Q+ -> LinkIn.buf_FFacktoFFack.val+ < LinkIn.postdrain+
   --    LinkIn.FFack.Q- -> LinkIn.buf_FFacktoFFack.val- < LinkIn.postdrain+
   VAR
     Click-RT2a : process rt (LinkIn.FFack.Q,  LinkIn.buf_FFacktoFFack.val,  GREEN);
     Click-RT2b : process rt (!LinkIn.FFack.Q, !LinkIn.buf_FFacktoFFack.val, GREEN);
   DEFINE
     LinkIn_stoprise_postdrain := Click-RT2a.stop | Click-RT2b.stop;
   --
   --(3) Click-RT3:
   --    ENV_fill+ (i.e., LinkIn.fill+)
   --    -> LinkIn.ImpliedJointIn.andin2-
   --       i.e., here propagation stops at LinkIn.postfill-
   --    <  LinkIn.buf_FFacktoackfar+
   --
   --    drain+ -> thisJoint.andin1- < FFreqtoreqfar+
   VAR
      Click-RT3a: process rt (ENV_fill, !LinkIn.postfill, GREEN);
      Click-RT3b: process rt (drain, !thisJoint.andin1, GREEN);
   DEFINE
      LinkIn_stop_xnorinfar := Click-RT3a.stop;
      LinkIn_stop_xorinfar  := Click-RT3b.stop;



   --RT constraints for LinkOut (eGasP)
   --
   --(1) eGasP-RT1:
   --    fill+  -> LinkOut.swtokeepHI+ < LinkOut.PGfill-
   --    ENV_drain+ -> LinkOut.swtokeepLO- < LinkOut.PGdrain-
   VAR
      eGasP-RT1a: process rt (fill,  LinkOut.swtokeepHI,  GREEN);
      eGasP-RT1b: process rt (ENV_drain, !LinkOut.swtokeepLO, GREEN);
   --
   --(2) eGasP-RT2:
   --    fill+  -> LinkOut.swtokeepLO+ < LinkOut.PGfill-
   --    ENV_drain+ -> LinkOut.swtokeepHI- < LinkOut.PGdrain-
   VAR
      eGasP-RT2a: process rt (fill,  LinkOut.swtokeepLO,  GREEN);
      eGasP-RT2b: process rt (ENV_drain, !LinkOut.swtokeepHI, GREEN);
   --
   --(3) eGasP-RT3:
   --    fill+
```

```
133      --    -> LinkOut.PGfill-, (no SW fight)
134      --       LinkOut. postfill-, (disable Cfill before buf_swtoCfill.val+)
135      --       thisJoint.andin2-, (disable EMPTY influence in Joint)
136      --    <  LinkOut.PGdrain+
137      --
138      --    ENV_drain+
139      --    -> LinkOut.PGdrain-, (no SW fight)
140      --       LinkOut.postdrain-, (disable Cdrain before buf_swtoCdrain.val+)
141      --       ImpliedJointOut.andin1- (i.e. here, LinkOut.postdrain-)
142      --    <  LinkOut.PGfill+
143      VAR
144         eGasP-RT3a1: process rt (fill, !LinkOut.PGfill,   GREEN);
145         eGasP-RT3a2: process rt (fill, !LinkOut.postfill, GREEN);
146         eGasP-RT3a3: process rt (fill, !thisJoint.andin2, GREEN);
147         --
148         eGasP-RT3b1: process rt (ENV_drain, !LinkOut.PGdrain,   GREEN);
149         eGasP-RT3b2: process rt (ENV_drain, !LinkOut.postdrain, GREEN);
150      --
151      --(4) eGasP-RT4:
152      --    fill+      -> LinkOut.Cfill-  < LinkOut.PGfill-
153      --    ENV_drain+ -> LinkOut.Cdrain- < LinkOut.PGdrain-
154      VAR
155         eGasP-RT4a: process rt (fill,      !LinkOut.Cfill,  GREEN);
156         eGasP-RT4b: process rt (ENV_drain, !LinkOut.Cdrain, GREEN);
157      --
158      DEFINE
159         LinkOut_stoprise_PGfill  := eGasP-RT3b1.stop | eGasP-RT3b2.stop;
160         LinkOut_stoprise_PGdrain := eGasP-RT3a1.stop | eGasP-RT3a2.stop |
            eGasP-RT3a3.stop;
161         --
162         LinkOut_stopfall_PGfill  := eGasP-RT1a.stop | eGasP-RT2a.stop | eGasP-RT4a.stop;
163         LinkOut_stopfall_PGdrain := eGasP-RT1b.stop | eGasP-RT2b.stop |
            eGasP-RT4b.stop;
164   -- END MODULE main
165   ----------------END MAIN PROGRAM Click-Joint-eGasP
166
```

```
Wall-clock execution time: 5 hours 49 minutes
############################################################################
system diameter: 143
reachable states: 1.32923e+006 (2^20.3422) out of 1.74085e+026 (2^87.1699)
############################################################################



*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 14:15:02 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
*** Copyright (c) 2010, Fondazione Bruno Kessler
*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson
WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG semimodular IN thisJoint.buf_postfull is true
-- specification AG semimodular IN thisJoint.buf_postempty is true
-- specification AG semimodular IN thisJoint.and is true
-- specification AG semimodular IN thisJoint.buf_postand is true
-- specification AG semimodular IN LinkIn.buf_postfill is true
-- specification AG semimodular IN LinkIn.buf_postdrain is true
-- specification AG semimodular IN LinkIn.xnor_gate is true
-- specification AG semimodular IN LinkIn.xor_gate is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoFFreq is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqnear is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqfar is true
-- specification AG semimodular IN LinkIn.buf_FFacktoFFack is true
-- specification AG semimodular IN LinkIn.buf_FFacktoacknear is true
-- specification AG semimodular IN LinkIn.buf_FFacktoackfar is true
-- specification AG semimodular IN LinkOut.buf_postfill is true
-- specification AG semimodular IN LinkOut.C_fill is true
-- specification AG !(set & reset) IN LinkOut.C_fill is true
-- specification AG semimodular IN LinkOut.PG_fill is true
-- specification AG ((set & val) -> A [ val U !set ] ) IN LinkOut.PG_fill is true
-- specification AG semimodular IN LinkOut.buf_postdrain is true
-- specification AG semimodular IN LinkOut.C_drain is true
-- specification AG !(set & reset) IN LinkOut.C_drain is true
-- specification AG semimodular IN LinkOut.PG_drain is true
-- specification AG ((set & val) -> A [ val U !set ] ) IN LinkOut.PG_drain is true
-- specification AG !(inHI & inLO) IN LinkOut.driveHIkeepLO is true
-- specification AG !(inHI & inLO) IN LinkOut.driveLOkeepHI is true
-- specification AG noFight IN LinkOut.statewire is true
-- specification AG noFloat IN LinkOut.statewire is true
-- specification AG semimodular IN LinkOut.buf_swtokeepLO is true
-- specification AG semimodular IN LinkOut.buf_swtokeepHI is true
-- specification AG semimodular IN LinkOut.buf_swtoEMPTY is true
-- specification AG semimodular IN LinkOut.buf_swtoCfill is true
-- specification AG semimodular IN LinkOut.buf_swtoFULL is true
-- specification AG semimodular IN LinkOut.buf_swtoCdrain is true
-- specification AG semimodular IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVIn is true
-- specification AG semimodular IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVOut is true
-- specification AG (fill -> EF !fill)  is true
-- specification AG (!fill -> EF fill)  is true
-- specification AG (drain -> EF !drain)  is true
-- specification AG (!drain -> EF drain)  is true

```

Appendix C

NuSMV Implementation and Results of Full FIFO of Fig. 8: The Mixed Family FIFO Model

"Click Link - Joint with C-element - eGasP Link"

This Appendix consists of the following components:

```
1  ------------------BEGIN LIBRARY EXTENSION JointwC
2  --BEGIN NuSMV Library
3  --Allie JointwCextension
4  --Honors Thesis May 2020
5
6
7  -- Library modules in here are formally instantiated as follows:
8  --   MODULE JointwC (FULL, EMPTY, stopfall_fire, stoprise_andin)
9
10
11 MODULE JointwC (FULL, EMPTY, stoprise_andin)
12 --FULL   : Boolean from input Link
13 --EMPTY  : Boolean from output Link
14 --fill   : Boolean to output Link
15 --drain  : Boolean to input Link
16 --stop*  : Boolean imported RT constraint stops
17 --
18 --NOTE:
19 --   assume that everything is reset initially, no new communications have come in yet
20 --
21    VAR
22       buf_postfull  : process cgate (FULL,  FALSE, stoprise_andin, FALSE);
23       buf_postempty : process cgate (EMPTY, FALSE, stoprise_andin, FALSE);
24       Celt          : process C (andin1 & andin2,!andin1 & !andin2, FALSE, FALSE, FALSE);
25       buf_postC     : process cgate (Celt.val, FALSE, FALSE, FALSE);
26    DEFINE
27       andin1 := buf_postfull.val;
28       andin2 := buf_postempty.val;
29       fire   := buf_postC.val;
30       fill   := fire;
31       drain  := fire;
32    --PROPERTIES
33      --safety
34        --semimodularity is checked within each cgate and within C
35      --progress
36        --every module instance is selected for evaluation within finite time
37        FAIRNESS running
38 --END MODULE JointwC
39 ------------------END LIBRARY EXTENSION JointwC
40
41
```

```
----------------BEGIN MAIN PROGRAM Click-JointwC-eGasP
MODULE main
   VAR
      thisJoint : process JointwC
                  (FULL, EMPTY, JointwC_stoprise_andin);
                  --starts reset

      LinkIn    : process Link_Click
                  (ENV_fill, drain, TRUE,
                    LinkIn_stoprise_postfill, LinkIn_stoprise_postdrain,
                    LinkIn_stop_xnorinfar, LinkIn_stop_xorinfar);
                   --starts FULL

      LinkOut   : process Link_eGasP
                  (fill, ENV_drain, FALSE,
                   LinkOut_stoprise_PGfill, LinkOut_stoprise_PGdrain,
                   LinkOut_stopfall_PGfill, LinkOut_stopfall_PGdrain);
                   --starts EMPTY

      ENVIn     : process cgate_lazyHI
                  (ENV_EMPTY, FALSE, FALSE, FALSE);
                   --starts reset

      ENVOut    : process cgate_lazyHI
                  (ENV_FULL, FALSE, FALSE, FALSE);
                   --starts reset
   DEFINE
      fire      := thisJoint.fire;
      fill      := thisJoint.fill;
      drain     := thisJoint.drain;
      FULL      := LinkIn.FULL;
      EMPTY     := LinkOut.EMPTY;
      ENV_FULL  := LinkOut.FULL;
      ENV_EMPTY := LinkIn.EMPTY;
      ENV_fill  := ENVIn.val;
      ENV_drain := ENVOut.val;

   --PROPERTIES
      --safety
        --Semimodularity  and other properties are checked within each process
      --progress
        --every module instance is selected for evaluation within finite time
        FAIRNESS running
        --there's real action!
        CTLSPEC AG (fill   -> (EF !fill));
        CTLSPEC AG (!fill  -> (EF fill));
        CTLSPEC AG (drain  -> (EF !drain));
        CTLSPEC AG (!drain -> (EF drain));



   --RT constaints for thisJoint:
   --
   --(1) JointwC-RT1: satisfied now by design
   --    fire+ -> !EMPTY, !FULL < fire-
   --
   --(2) JointwC-RT2: simplified by design
   --    fire+
   --    -> LinkIn.postdrain-, LinkOut.postfill-
   --    <  thisJoint.andin1+, thisJoint.andin2+
   VAR
      JointwC-RT2c: process rt (fire, !LinkIn.postdrain, GREEN);
      JointwC-RT2d: process rt (fire, !LinkOut.postfill, GREEN);
   DEFINE
      JointwC_stoprise_andin := JointwC-RT2c.stop | JointwC-RT2d.stop;

```

```
67
68
69      --RT constraints for LinkIn (Click):
70      --
71      --(1) Click-RT1:
72      --    LinkIn.FFreq.Q+ -> LinkIn.buf_FFreqtoFFreq.val+ < LinkIn.postfill+
73      --    LinkIn.FFreq.Q- -> LinkIn.buf_FFreqtoFFreq.val- < LinkIn.postfill+
74      VAR
75        Click-RT1a : process rt (LinkIn.FFreq.Q,  LinkIn.buf_FFreqtoFFreq.val,  GREEN);
76        Click-RT1b : process rt (!LinkIn.FFreq.Q, !LinkIn.buf_FFreqtoFFreq.val, GREEN);
77      DEFINE
78        LinkIn_stoprise_postfill := Click-RT1a.stop | Click-RT1b.stop;
79      --
80      --(2) Click-RT2:
81      --    LinkIn.FFack.Q+ -> LinkIn.buf_FFacktoFFack.val+ < LinkIn.postdrain+
82      --    LinkIn.FFack.Q- -> LinkIn.buf_FFacktoFFack.val- < LinkIn.postdrain+
83      VAR
84        Click-RT2a : process rt (LinkIn.FFack.Q,  LinkIn.buf_FFacktoFFack.val,  GREEN);
85        Click-RT2b : process rt (!LinkIn.FFack.Q, !LinkIn.buf_FFacktoFFack.val, GREEN);
86      DEFINE
87        LinkIn_stoprise_postdrain := Click-RT2a.stop | Click-RT2b.stop;
88      --
89      --(3) Click-RT3:
90      --    ENV_fill+ (i.e., LinkIn.fill+)
91      --    -> LinkIn.ImpliedJointIn.andin2-
92      --       i.e., here propagation stops at LinkIn.postfill-
93      --    <  LinkIn.buf_FFacktoackfar+
94      --
95      --    drain+ -> thisJoint.andin1- < FFreqtoreqfar+
96      VAR
97         Click-RT3a: process rt (ENV_fill, !LinkIn.postfill, GREEN);
98         Click-RT3b: process rt (drain, !thisJoint.andin1, GREEN);
99      DEFINE
100        LinkIn_stop_xnorinfar := Click-RT3a.stop;
101        LinkIn_stop_xorinfar  := Click-RT3b.stop;
102
103
104
105     --RT constraints for LinkOut (eGasP)
106     --
107     --(1) eGasP-RT1:
108     --    fill+  -> LinkOut.swtokeepHI+ < LinkOut.PGfill-
109     --    ENV_drain+ -> LinkOut.swtokeepLO- < LinkOut.PGdrain-
110     VAR
111        eGasP-RT1a: process rt (fill,  LinkOut.swtokeepHI,  GREEN);
112        eGasP-RT1b: process rt (ENV_drain, !LinkOut.swtokeepLO, GREEN);
113     --
114     --(2) eGasP-RT2:
115     --    fill+  -> LinkOut.swtokeepLO+ < LinkOut.PGfill-
116     --    ENV_drain+ -> LinkOut.swtokeepHI- < LinkOut.PGdrain-
117     VAR
118        eGasP-RT2a: process rt (fill,  LinkOut.swtokeepLO,  GREEN);
119        eGasP-RT2b: process rt (ENV_drain, !LinkOut.swtokeepHI, GREEN);
120     --
121     --(3) eGasP-RT3:
122     --    fill+
123     --    -> LinkOut.PGfill-, (no SW fight)
124     --       LinkOut. postfill-, (disable Cfill before buf_swtoCfill.val+)
125     --       thisJoint.andin2-, (disable EMPTY influence in Joint)
126     --    <  LinkOut.PGdrain+
127     --
128     --    ENV_drain+
129     --    -> LinkOut.PGdrain-, (no SW fight)
130     --       LinkOut.postdrain-, (disable Cdrain before buf_swtoCdrain.val+)
131     --       ImpliedJointOut.andin1- (i.e. here, LinkOut.postdrain-)
132     --    <  LinkOut.PGfill+
```

```
133     VAR
134        eGasP-RT3a1: process rt (fill, !LinkOut.PGfill,   GREEN);
135        eGasP-RT3a2: process rt (fill, !LinkOut.postfill, GREEN);
136        eGasP-RT3a3: process rt (fill, !thisJoint.andin2, GREEN);
137        --
138        eGasP-RT3b1: process rt (ENV_drain, !LinkOut.PGdrain,   GREEN);
139        eGasP-RT3b2: process rt (ENV_drain, !LinkOut.postdrain, GREEN);
140     --
141     --(4) eGasP-RT4:
142     --    fill+      -> LinkOut.Cfill-  < LinkOut.PGfill-
143     --    ENV_drain+ -> LinkOut.Cdrain- < LinkOut.PGdrain-
144     VAR
145        eGasP-RT4a: process rt (fill,      !LinkOut.Cfill,  GREEN);
146        eGasP-RT4b: process rt (ENV_drain, !LinkOut.Cdrain, GREEN);
147     --
148     DEFINE
149        LinkOut_stoprise_PGfill  := eGasP-RT3b1.stop | eGasP-RT3b2.stop;
150        LinkOut_stoprise_PGdrain := eGasP-RT3a1.stop | eGasP-RT3a2.stop |
           eGasP-RT3a3.stop;
151        --
152        LinkOut_stopfall_PGfill  := eGasP-RT1a.stop | eGasP-RT2a.stop | eGasP-RT4a.stop;
153        LinkOut_stopfall_PGdrain := eGasP-RT1b.stop | eGasP-RT2b.stop |
           eGasP-RT4b.stop;
154  -- END MODULE main
155  ----------------END MAIN PROGRAM Click-JointwC-eGasP
156
```

```
Wall-clock execution time: 2 hours 25 minutes
############################################################################
system diameter: 143
reachable states: 851328 (2^19.6994) out of 1.08803e+025 (2^83.1699)
############################################################################



*** This is NuSMV 2.5.4 (compiled on Fri Oct 28 14:15:02 UTC 2011)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
*** Copyright (c) 2010, Fondazione Bruno Kessler
*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson
WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG semimodular IN thisJoint.buf_postfull is true
-- specification AG semimodular IN thisJoint.buf_postempty is true
-- specification AG semimodular IN thisJoint.Celt is true
-- specification AG !(set & reset) IN thisJoint.Celt is true
-- specification AG semimodular IN thisJoint.buf_postC is true
-- specification AG semimodular IN LinkIn.buf_postfill is true
-- specification AG semimodular IN LinkIn.buf_postdrain is true
-- specification AG semimodular IN LinkIn.xnor_gate is true
-- specification AG semimodular IN LinkIn.xor_gate is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoFFreq is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqnear is true
-- specification AG semimodular IN LinkIn.buf_FFreqtoreqfar is true
-- specification AG semimodular IN LinkIn.buf_FFacktoFFack is true
-- specification AG semimodular IN LinkIn.buf_FFacktoacknear is true
-- specification AG semimodular IN LinkIn.buf_FFacktoackfar is true
-- specification AG semimodular IN LinkOut.buf_postfill is true
-- specification AG semimodular IN LinkOut.C_fill is true
-- specification AG !(set & reset) IN LinkOut.C_fill is true
-- specification AG semimodular IN LinkOut.PG_fill is true
-- specification AG ((set & val) -> A [ val U !set ] ) IN LinkOut.PG_fill is true
-- specification AG semimodular IN LinkOut.buf_postdrain is true
-- specification AG semimodular IN LinkOut.C_drain is true
-- specification AG !(set & reset) IN LinkOut.C_drain is true
-- specification AG semimodular IN LinkOut.PG_drain is true
-- specification AG ((set & val) -> A [ val U !set ] ) IN LinkOut.PG_drain is true
-- specification AG !(inHI & inLO) IN LinkOut.driveHIkeepLO is true
-- specification AG !(inHI & inLO) IN LinkOut.driveLOkeepHI is true
-- specification AG noFight IN LinkOut.statewire is true
-- specification AG noFloat IN LinkOut.statewire is true
-- specification AG semimodular IN LinkOut.buf_swtokeepLO is true
-- specification AG semimodular IN LinkOut.buf_swtokeepHI is true
-- specification AG semimodular IN LinkOut.buf_swtoEMPTY is true
-- specification AG semimodular IN LinkOut.buf_swtoCfill is true
-- specification AG semimodular IN LinkOut.buf_swtoFULL is true
-- specification AG semimodular IN LinkOut.buf_swtoCdrain is true
-- specification AG semimodular IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVIn is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVIn is true
-- specification AG semimodular IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX val) IN ENVOut is true
-- specification AG (((!stop_rise & set) & !val) -> EX !val) IN ENVOut is true
-- specification AG (fill -> EF !fill)  is true
-- specification AG (!fill -> EF fill)  is true
-- specification AG (drain -> EF !drain)  is true
-- specification AG (!drain -> EF drain)  is true
```