Flexible Compilation and Refinement of Asynchronous Circuits

Ebelechukwu Esimai and Marly Roncken
Asynchronous Research Center and Computer Science Department
Maseeh College of Engineering and Computer Science, Portland State University, Portland, Oregon, USA
esimai@pdx.edu, mroncken@pdx.edu

***Abstract*—We present compilation and refinement techniques for translating parallel programs with message passing into asynchronous circuits. Instead of compiling programs directly into circuits using a fixed protocol and circuit family — as is traditionally done — we compile programs into a circuit-neutral model consisting of communication channels with storage, called *Links*, and storage-free computation modules, called *Joints*. We refine this model into a gate-level circuit by reducing storage and selecting protocols and circuit families. The final circuits combine 2- and 4-phase protocols and various circuit families. We give two refinement examples. The first refinement safely removes data storage from Links to improve circuit area and power. The second refinement safely splits atomic Joint actions to improve circuit analysis. Both refinements introduce 4-phase protocols for which we give a formal Link-Joint model and circuits in Click, Set-Reset, and GasP. We are implementing this *compile then refine* approach as a shallow embedding in an open-source design flow.**

***Index Terms*—asynchronous circuits, communication protocols, design automation, compilation, formal models, refinement**

I. Introduction

This paper addresses design automation for asynchronous circuits. Fig. 1 provides an overview of the design flow. The Links and Joints in the middle provide an abstraction level between algorithmic programs and electronic circuits. Proposed in 2015, Links and Joints are silent about protocols and circuit families, and bind circuit decisions late to increase design and test flexibility [20]. But until now, Links and Joints could be used only for small or regular circuits, and by experts only. The embedding in Fig. 1 makes Links and Joints accessible for general use in large electronic systems.

As design flow we selected ACT [1], [13], [14], because it is open-source, in active use, supports programs with both data- and control-flow, and builds on in-depth knowledge in asynchronous circuit design and analysis.

The embedding in ACT lets us (1) compile ACT programs into Link-Joint networks, (2) refine networks by choosing protocols and circuit families, and (3) pass gate-level circuits back to ACT for further processing. We currently use a shallow embedding, leaving ACT in (1) and re-entering in (3).

The focus of this paper is on compilation and refinement. Sections II–IV provide background on compilation and Links and Joints. Fig. 2 shows Link-Joint library elements used by the compiler. Figs. 3–4 give simulated compilation examples. Sections V–VI and Figs. 5–8 discuss two refinements. *This paper is theoretical and meant to be used as reference paper.*

This work is supported in part by private sponsors through the Portland State University Foundation and in part by the Mayo Clinic under subcontract SPPDG-052 for "Computing Systems Based on the Link-Joint Paradigm."

II. Compilation

Our compilation strategy is based on a large body of work, both past [2]–[7], [15], [16] and present [1], [13], [14], with an impressive variety of asynchronous designs, chips, and market products. All start with a message-passing parallel programming language based on Hoare's CSP [11]. Using syntax-directed compilation, each CSP program is translated into a network of *handshake circuits* connected by *channels*. The handshake protocol and circuit family are determined prior to compilation.

Our compiler is branched off Yale's asynchronous circuit toolkit (ACT) [14]. We compile ACT programs with data-flow parts written in ACT sublanguage *dataflow* and control-flow parts in CHP (*Communicating Hardware Processes*) [13]. But instead of generating networks of handshake circuits, we generate circuit-neutral Link-Joint networks that work for a variety of protocols and circuit families. The key challenge in developing this compiler was to find a representation of handshake circuits in terms of Links and Joints. Where handshake circuits compute *and* store state (using their channels merely to transport state information), in our case, the Joints compute but the Links store. Once we had a systematic representation, we could more or less re-use Yale's compiler.

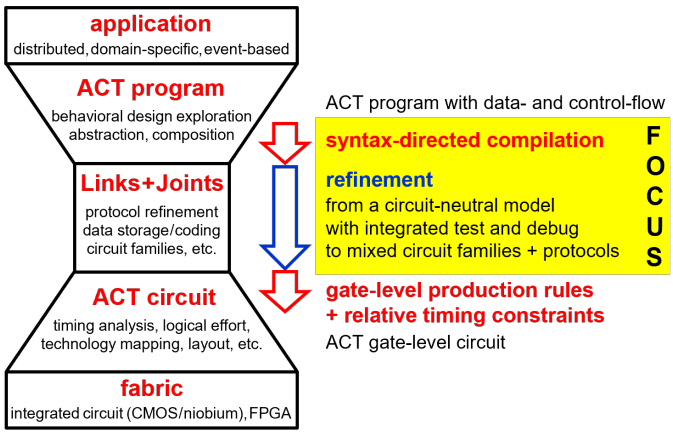

Fig. 1: ACT electronic design automation framework to design and implement distributed, domain-specific, event-based systems as FPGA or integrated circuits [1], [13], [14]. The hourglass, originally proposed by Kees van Berkel [4], emphasizes the separation between program and circuit concerns. This paper focuses on the middle part of the hourglass. We compile ACT programs into Links and Joints, a circuit-neutral model with integrated test and debug [17]–[20], which we translate into gate-level circuits through stepwise refinement. Final circuits may combine different protocols and circuit families.

979-8-3503-0572-2/23/$31.00 ©2023 IEEE

III. Links and Joints

Links and Joints separate *states* from *actions*. Where Links transport and store state information, the stateless Joints compute and control the flow of information. In essence, each Joint is a place where Links meet to exchange information.

We use Link-Joint networks to provide an abstraction level between algorithmic programs and electronic circuits. Each network alternates Links and Joints, using port connections. Links have two ports, called *A* and *B*. Joints may have several ports, with formal port names defined only in the scope of the Joint and used in its formal specification. Initial Link-Joint specifications are *sufficiently detailed* to facilitate transparent translations into circuits and *sufficiently abstract* to embrace circuit implementations with different protocols, logic signaling and data representations, circuit families, and fabrication means. Later refinements tailor the Links and Joints to fit a particular circuit purpose or preference — see Sections V–VI.

Section III-A provides a formal Link-Joint model. Sections III-B to III-I specify the Joints in Fig. 2 and explain their role in the example program compilations in Figs. 3–4.

A. A Circuit-Neutral Model with Integrated Test and Debug

We use a shared variable model [9] and guarded command specifications [8] to express Links, Joints, their behaviors, and their port interactions.

Each Link has two ports, *A* and *B*, and shares three variables, *turn*, $data_{\text{AtoB}}$, and $data_{\text{BtoA}}$ with the Joint ports connected to *A* and *B*. Link variables $data_{\text{AtoB}}$ and $data_{\text{BtoA}}$ contain zero or more bits of data going from *A* to *B* and *B* to *A*, respectively. Following good conversation practice, the Joint ports take turns updating the Link variables, including variable *turn*. Each Link keeps track of whose turn it is, *A*'s or *B*'s, and shares this information with both Joints via Link variable *turn*.

We specify each Joint action as a guarded command, using the following terminology for Joint ports p, p_1, p_2.

- Boolean *myturn(p)* is *true* if and only if p has the turn.
- Data *myR(p)* and *myW(p)* are data read and written by p — going from Link variables to p and vice versa.
- Assignment *yourturn(p)* changes Link variable *turn* so that *myturn(p)* becomes *false* and $myturn(p_{\text{peer}})$ becomes *true*, where p_{peer} is *A* if p connects to *B*, otherwise *B*.
- We use $myturn(p_1, p_2)$ for $myturn(p_1) \wedge myturn(p_2)$, and $yourturn(p_1, p_2)$ for $yourturn(p_1); yourturn(p_2)$, etc.

This terminology allows the specifications to be silent as to whether Joint port p connects to Link port *A* or Link port *B*.

Each guarded command is of the form $guard \rightarrow command$, where *guard* is a Boolean expression and *command* a sequence of assignments. We may use $guard_1 \rightarrow guard_2 \rightarrow command$ as an alternative notation for $guard_1 \wedge guard_2 \rightarrow command$. Guarded commands execute atomically, in mutual exclusion, and only when their guard is valid [8].

For initialization, test, and debug we can control and observe Link variables externally. To avoid Joint interference, we can stop any or all Joint action. For this purpose, each Joint has its own (arbitrated) *go* signal [20]: *go* permits and $\neg go$ denies command execution. Correspondingly, each guarded command includes a permissive *go* signal in its guard.

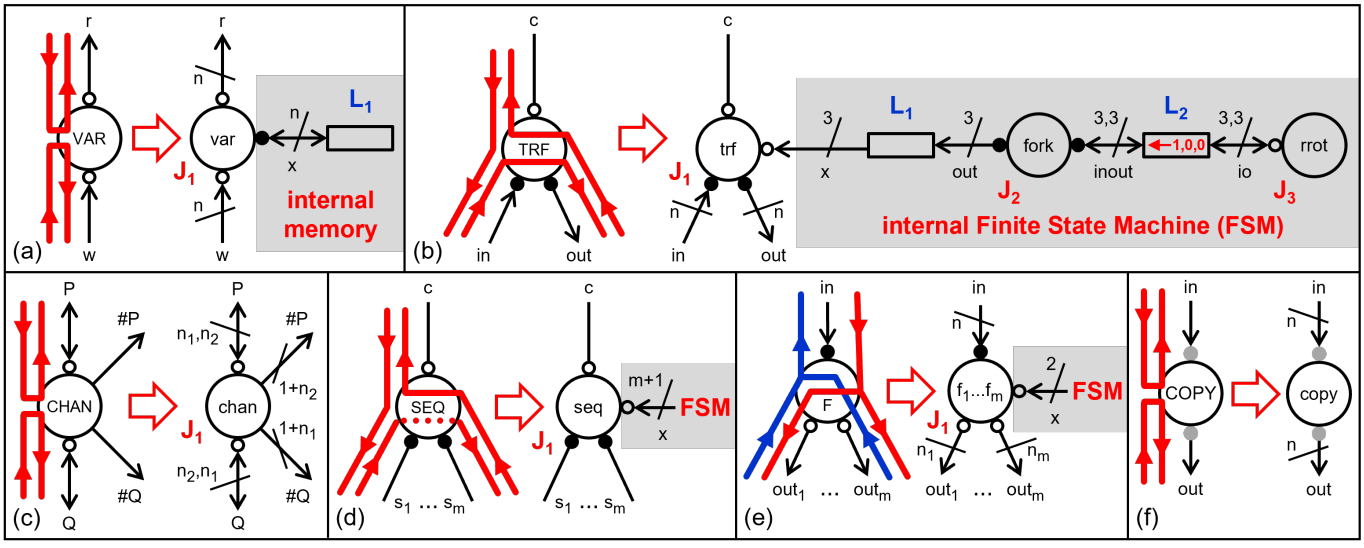

Fig. 2: Joint library elements used by the compiler. We draw each Joint as a big circle containing the Joint name, and attach its ports as small circles with a formal port name. Basic Joints (c,f) have lowercase names. Composite Joints (a,b,d,e) have uppercase names. We omit *drawing* internal ports of composite Joint — but the port connections are still there. By default, all Joint ports are colored grey to indicate that connecting Links can be initialized freely [9]. When we color a port black or white, we merely indicate a specific initial Link state for normal operation of the Joint in a given context — e.g., in the context of ACT-CHP program compilation — but we retain freedom of initialization for other scenarios. For readability, we omit drawing the individual and external *go* signals for each Joint. We draw a Link as a rectangle, to indicate that it stores information. The pictures omit the Link port names. The flow of data in both, either, or neither port direction is indicated by the presence or absence of arrows. We marked the flow control for each Joint with fat line-arrows, colored in red and blue to separate paths. We will continue these markings to express the higher-level flow control in the compiled designs in Figs. 3–4.

B. Joint VAR

VAR in Fig. 2(a) represents an *n*-bit program variable, $n \geq 0$, and is used in Fig. 3(b). *VAR* has a basic Joint *var* and read and write ports *r*, *w* for mutual exclusive use. We store the variable's value in Link L_1 connected to internal *VAR* port *x*. *VAR* has two guarded commands, one for read, one for write. Neither relinquish the turn on *x*, which we can set externally in L_1 for initialization and test purposes.

myturn(r, x) $\wedge$ *go* $\rightarrow$ *myW(r)* := *myR(x)* ; *yourturn(r)*
myturn(w, x) $\wedge$ *go* $\rightarrow$ *myW(x)* := *myR(w)* ; *yourturn(w)*

C. Joint CHAN

Because they are storage-free, we represent CHP channels as Joints — *not* as Links. This makes perfect sense, because channels synchronize parallel processes and Joints excel at bringing together multiple participants by synchronizing them. Joint *CHAN* in Fig. 2(c) represents a channel with two ports, *P* and *Q*, for connecting two communicating processes that exchange n_1 data bits from P to Q, n_2 data bits from Q to P, where $n_1, n_2 \geq 0$.

Each process may probe the channel to sense if its partner is ready to communicate. Probe signals *#P*, *#Q* can be read and written directly, without a communication protocol. A process can probe communication readiness, using 1 bit, and the data sent by its partner — n_2 bits for *#P*, n_1 for *#Q*. Figs. 3(b) and 4(a) show designs with probe-less and readiness-only probes. Joint *CHAN* has one guarded command, specified as follows.

myturn(P, Q) $\wedge$ *go* $\rightarrow$
myW(P) := *myR(Q)* ; *myW(Q)* := *myR(P)* ; *yourturn(P, Q)*

D. Joint TRF

TRF in Fig. 2(b) is generated as part of a CHP assignment or communication. Fig. 3(b) shows a communication example. *TRF* has a basic Joint *trf*, and ports *c*, *in*, *out*. When prompted by *c*, *TRF* first requests *n* bits of data, $n \geq 0$, from variables or channels connected to port *in*, which it then "transfers" to variables or channels connected to port *out*, before reporting completion at *c*. This sequence is controlled by 1-hot 3-bit string *myR(x)*, generated by a Finite State Machine (FSM) connected to internal *TRF* port *x* and discussed in Section III-E. *TRF* executes three guarded commands, specified as follows.

myturn(c, in, out, x) $\wedge$ *go* $\rightarrow$
  myR(x)[0] $\rightarrow$ *yourturn(in, x)*
  myR(x)[1] $\rightarrow$ *myW(out)* := *myR(in)* ; *yourturn(out, x)*
  myR(x)[2] $\rightarrow$ *yourturn(c, x)*

E. Finite State Machine (FSM) and Joints fork and rrot

The FSM in Fig. 2(b) maintains a 1-hot bit string. Joint *TRF*, specified in Section III-D, uses the 1-hot bit position to decide which command to execute. With three guarded commands, *TRF* requires a 3-bit FSM string. After each *TRF* execution, the FSM right-rotates the bits by one position around the string. To simplify the connection at *TRF* port *x*, the FSM has two basic Joints, *fork* and *rrot*, a unidirectional Link L_1 from *fork* port *out* to *TRF* port *x*, and a bidirectional Link L_2 between *fork* port *inout* and *rrot* port *io*. We can initialize the FSM so that (1) the leftmost bit of the string stored in L_2 in the direction from *rrot* to *fork* is 1-hot, and (2) *fork* has the turn on both its ports and is ready to execute its command.

(*fork*)
myturn(inout, out) $\wedge$ *go* $\rightarrow$
myW(inout, out) := *myR(inout)* ; *yourturn(inout, out)*

Joint *fork* copies the FSM string from L_2 to L_1 and L_2. Its executions alternate with those of *TRF* and *rrot*. Joint *rrot* right-rotates the FSM string and returns the result to *fork*.

(*rrot*)
myturn(io) $\wedge$ *go* $\rightarrow$ *myW(io)* := *rrot(myR(io))* ; *yourturn(io)*

F. Joint SEQ

SEQ in Fig. 2(d) represents sequential composition and is used in Figs. 3–4. It has basic Joint *seq*, startup port *c*, and ports s_1 to s_m for the *m*, $m \geq 1$, program statements it sequences when prompted by *c*. It uses an m+1-bit internal FSM port *x* — see Section III-E — to sequence its commands, which are specified as follows, using index *i*, $0 \leq i < m$.

myturn(c, $s_1..s_m$*, x)* $\wedge$ *go* $\wedge$ *myR(x)[i]* $\rightarrow$ *yourturn(*s_{i+1}*,x)*
myturn(c, $s_1..s_m$*, x)* $\wedge$ *go* $\wedge$ *myR(x)[m]* $\rightarrow$ *yourturn(c, x)*

G. Joint F — for Refinement in Section VI

Joint *F* in Fig. 2(e) provides an alternative solution for evaluating expressions. Rather than sending evaluated results over a single Link as *E*, $E_{\text{waitcycle}}$ in Figs. 3–4, *F* distributes them over multiple Links. We use *F* in Section VI as reference example for protocol refinement. *F* has input port *in* and output ports out_1 to out_m, $m \geq 1$. When prompted by its output ports, *F* first requests data from *in* to which it applies *m* functions f_i as output data for out_i. *F* uses a 2-bit internal FSM port *x* (see Section III-E) to sequence its two commands, as follows.

myturn(in, $out_1..out_m$*, x)* $\wedge$ *go* $\wedge$ *myR(x)[0]* $\rightarrow$
  yourturn(in, x)
myturn(in, $out_1..out_m$*, x)* $\wedge$ *go* $\wedge$ *myR(x)[1]* $\rightarrow$
  myW(out_1*)*:=f_1 *(myR(in))* ;
  ...
  myW(out_m*)*:=f_m *(myR(in))* ;
  yourturn($out_1..out_m$*, x)*

H. Joint COPY

COPY in Fig. 2(f) copies *n* bits of data, $n \geq 0$, from its input port, *in*, to its output port, *out*, and is used in Fig. 3(a). Joint *COPY* executes the following guarded command.

myturn(in,out) $\wedge$ *go* $\rightarrow$ *myW(out)* := *myR(in)* ; *yourturn(in,out)*

I. Probed Selection and Joints $E_{\text{waitcycle}}$, SEL_{nondet}, $WAIT_{\text{cycle}}$

CHP program *BRTbuffer* in Fig. 4 uses nondeterministic selection based on probe conditions. The two conditions are evaluated in $E_{\text{waitcycle}}$ Joint J_7, which waits until either or both are valid before requesting probe snapshots to provide stable evaluation results for selection by SEL_{nondet} Joint J_6. SEL_{nondet} never sees the probes, and is therefore less interesting for discussion here. Its specification can be found in Appendix-E.

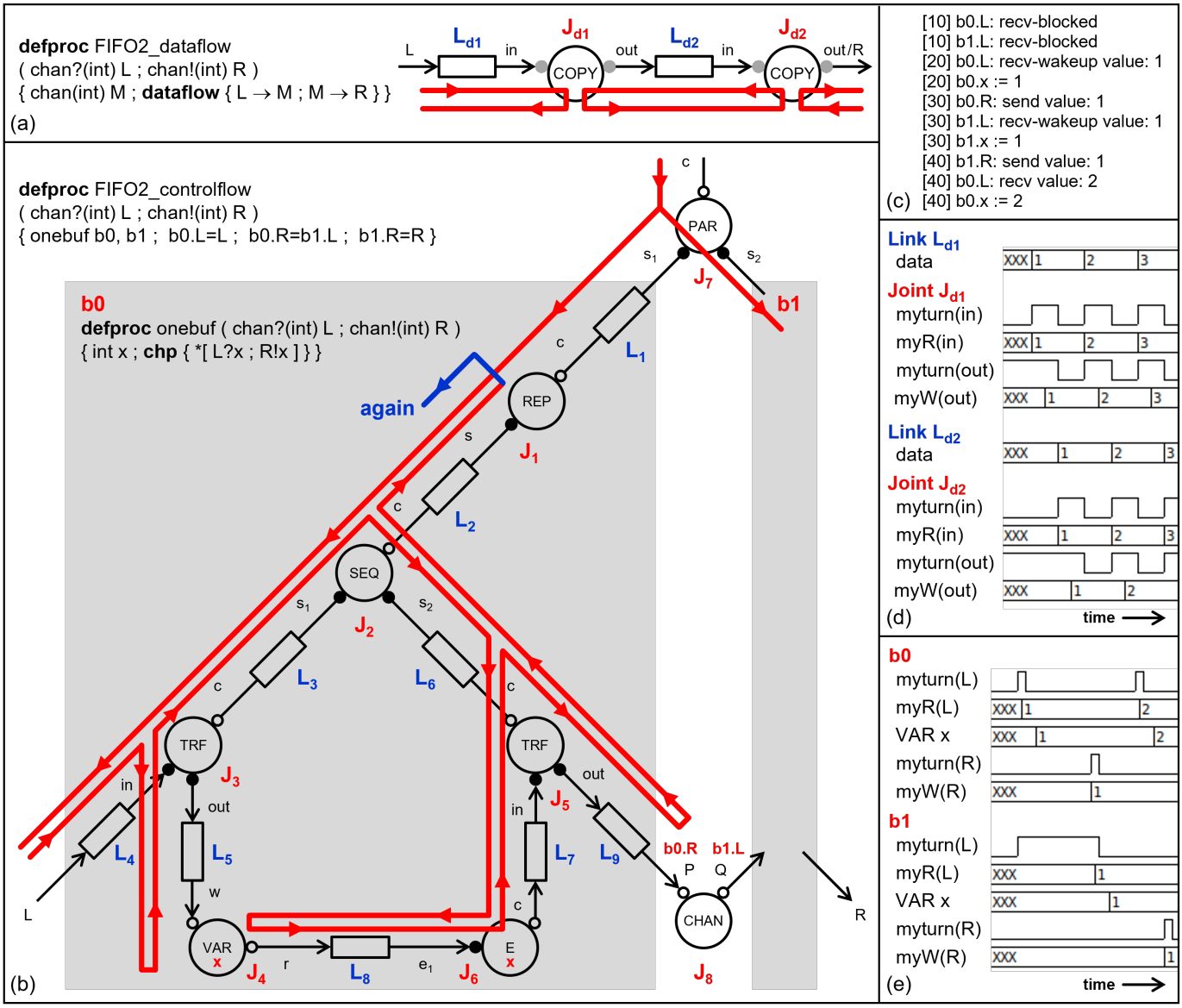

Fig. 3: ACT supports data- and control-flow programs [13]. Panels (a) and (b) give two different ACT programs for a linear first-in-first-out (FIFO) buffer that can store zero, one, or two data items. The programs have the same channel interface but different Link-Joint translations. Data enter at channel L and leave at channel R. The fat red or blue line-arrows mark the flow of control through each Link-Joint network.

(a) ACT *dataflow* program version and compiled Link-Joint network with two *COPY* Joints and two storage Links, in linear arrangement. We translate program statement $L \rightarrow M$ to (1) Link L_{d1} to store the data coming in on program channel L, and (2) Joint J_{d1} whose input port, *in*, connects to L_{d1} and who copies L_{d1}'s data to its output port, *out*, for program channel M. A similar translation for $M \rightarrow R$ generates Link L_{d2}, to store the data for M, and Joint J_{d2} who copies L_{d2}'s data to its output port *out* for program channel R.

(b) ACT hierarchical control-flow version and compiled Link-Joint network. Top-level program, *FIFO2_controlflow*, creates two *onebuf* instances, *b0* and *b1*, and serially connects their channels. We compile *FIFO2_controlflow* into (1) Link-Joint network instances for *b0* and *b1*, (2) *PAR* Joint J_7 to execute *b0* and *b1* in parallel, and (3) *CHAN* Joint J_8 to combine *b0.R* and *b1.L* into a single channel. Process *onebuf*, programmed in ACT sublanguage CHP can buffer zero or one data items. We compile *onebuf* by following its syntactic structure. Given that the entire program, *[L?x ; R!x], starts with repetition "*", we generate a repetition Joint: *REP* Joint J_1. The repeated program fragment, L?x ; R!x, is sequential, as indicated by the sequence operator ";", and so we generate a sequencing Joint, *SEQ* Joint J_2, and connect it to *REP* using Link L_2. The first of the two sequenced program statements, L?x, is a communication input over *onebuf* channel L, which requires synchronization with a communication output over L by a parallel process. The value sent by the parallel process is stored in *onebuf* variable x. We compile L?x to (1) *TRF* Joint J_3, (2) *VAR* Joint J_4 for x, (3) Links L_4 and L_5 to connect L to x via Joint *TRF*, and (4) Link L_3 to connect this translation to *SEQ* port s_1. The second program statement, R!x, is a communication output over *onebuf* channel R, which compiles to (1) *TRF* Joint J_5, (2) *E* Joint J_6 for output expression "x," (3) Links L_7, L_8, L_9 to connect variable x to *E* and *b0.R* via Joint *TRF*, and (4) Link L_6 to connect this translation to *SEQ* port s_2.

(c) ACT program simulation results for *FIFO2_controlflow* in (b), timed top to bottom, showing two inputs, one handover, and one output.

(d) Verilog waveforms for the Link-Joint network in (a), timed left to right, showing three inputs and internal handovers, and two outputs.

(e) Verilog waveforms for the Link-Joint network in (b), timed left to right, showing two inputs, one internal handover, and one output.

In Fig. 4, probes *#p*, *#g* come into Joint $E_{\text{waitcycle}}$ as raw signals with the formal names p_1, p_2. Their stable snapshots come in as protocol signals $myR(e_1)$, $myR(e_2)$ at ports e_1, e_2. $E_{\text{waitcycle}}$, with startup port *c*, uses an internal 2-bit FSM port *x* (see Section III-E) to sequence two guarded commands, which are specified using probe conditions $g[1]=p_1$, $g[2]=p_2$.

// ($E_{\text{waitcycle}}$) await valid condition before taking snapshots
$myturn(c, e_1, e_2, x) \wedge go \wedge myR(x)[0] \wedge (g[1] \vee g[2]) \rightarrow$
$yourturn(e_1, e_2, x)$
// return stable evaluation results
$myturn(c, e_1, e_2, x) \wedge go \wedge myR(x)[1] \rightarrow$
$myW(c) := g[1,2](myR(e_1, e_2) \,/\, p_1, p_2)$; $yourturn(c, x)$

The conditions can be extended with program variables by adding one more FSM bit and one more guarded command to request the variable values prior to awaiting a valid condition.

Stable probe snapshots are provided by Joint $WAIT_{cycle}$. Fig. 4 shows its instantiation, J_8, for probe *#p*. $WAIT_{cycle}$, with startup port *r*, uses an internal arbiter and 2-bit FSM port *x* to arbitrate the raw probe input against $myturn(x)$ for a duration of one FSM cycle. $WAIT_{cycle}$ returns a bit value, never a probe. Its guarded command specification uses mutual exclusive arbiter results $(grant_{\text{probe}}, grant_x) = arbiter(probe, myturn(x))$.

// ($WAIT_{cycle}$) arbitrate for one FSM cycle
$myturn(r, x) \wedge go \wedge myR(x)[0] \rightarrow yourturn(x)$
// return stable probe snapshot
$myturn(r, x) \wedge go \wedge myR(x)[1] \rightarrow$
$myW(r) := (0$ if $grant_x$, 1 if $grant_{\text{probe}})$; $yourturn(r, x)$

IV. Summary of Sections II–III

To re-use large parts of the existing compiler strategies we had to find the right Joints — a non-trivial challenge! The examples in Fig. 3(a)–(b) show how we compile both data-flow and control-flow programs in a syntax-directed way. Our strategy for compiling data-flow programs is to store data before using data, but the alternative, use before store, works equally well. Note that Joints *fork* and *rrot* in the internal finite state machine (FSM) in Fig. 2(b) are data-flow Joints — they control the execution sequence of composite Joints like *TRF*.

Fig. 3 clearly shows "what you program is what you get." Know your application domain: Link-Joint network (b) is large because we solved a data-flow problem with a control-flow program. We validate our compilation by comparing program simulations in ACT to Link-Joint simulations in Verilog (c–e).

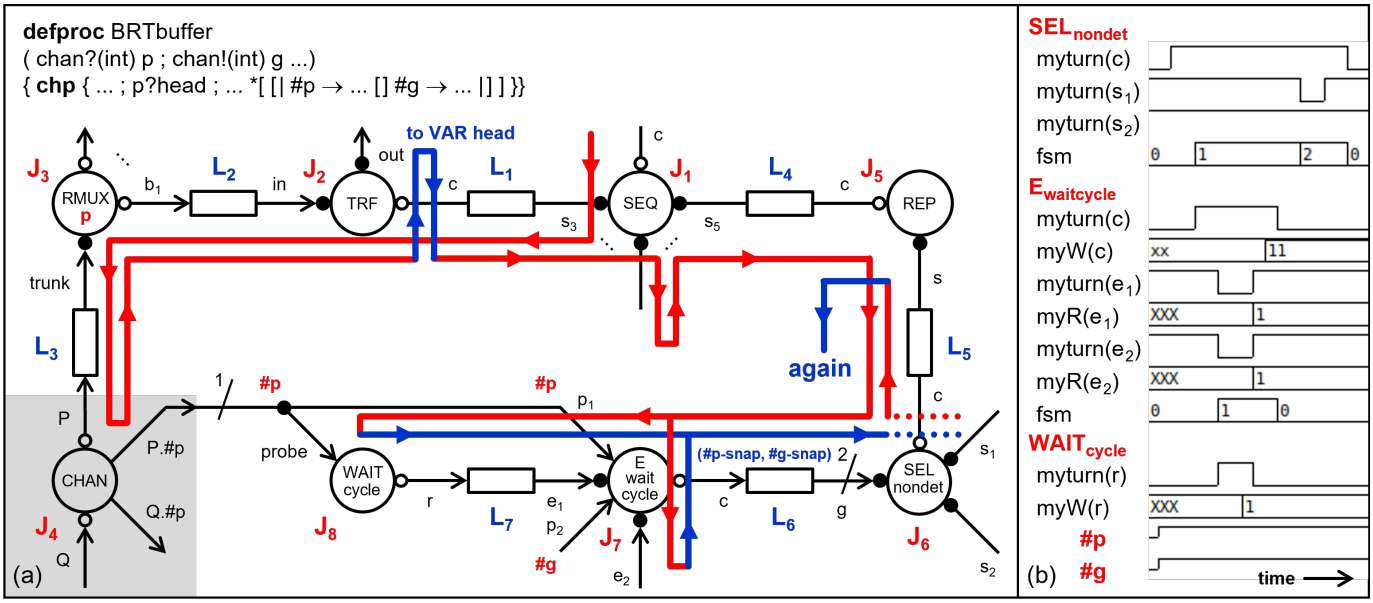

Fig. 4: ACT-CHP program fragment *BRTbuffer* and its corresponding Link-Joint network in (a) are part of a bounded response time buffer — a first-in-first-out buffer where the cycle time between inputs and outputs is bounded and independent of the buffer capacity. The program is based on the design by Kessels and Rem [12]. When the buffer is neither empty nor full it may receive both "put" (*p*) and "get" (*g*) communication requests from its environment. *BRTbuffer* serializes concurrent *p* and *g* requests. The program fragment focuses on the first *p* communication and the following (repeated) nondeterministic selection that uses probes to sense if the environment requests *p*, *g*, or both.

(a) The conditional select statement compiles into a network of Links and Joints including SEL_{nondet} for nondeterministic selection, $E_{\text{waitcycle}}$ for evaluating the probe conditions, and $WAIT_{\text{cycle}}$ for taking a stable probe snapshot, *#p-snap*, of probe *#p*. The various *p* communications in the program are multiplexed at *RMUX* Joint J_3. Note that *p* communications come in at *CHAN* J_4 port *P*, and *p* probes at *CHAN* wire connection *#p*. Because *BRTbuffer* probes for communication readiness without data, *#p* has bit width 1. The isolation of *CHAN* into a grey box hints at the fact that *CHAN* is compiled during parallel composition of *BRTbuffer*, as indicated and explained in Fig. 3(b).

(b) The Verilog simulation waveforms in (b) follow the fat red and blue path in the Link-Joint network from SEL_{nondet} to $E_{\text{waitcycle}}$, $WAIT_{\text{cycle}}$, and back in reverse direction. Both probes *#p* and *#g* have bit value 1 in this simulation fragment. As a result, $E_{\text{waitcycle}}$ receives bit value 1 for $myR(e_1)$ from $WAIT_{\text{cycle}}$ Joint J_8 and, ditto, bit value 1 for $myR(e_2)$ — which it combines into 2-bit string (1,1) for $myW(c)$, which arrives at SEL_{nondet} port *g*. Joint SEL_{nondet} favors *#p* here, and executes the corresponding s_1 statement. The *fsm* waveforms for SEL_{nondet} and $E_{\text{waitcycle}}$ indicate the 1-hot bit position in $myR(x)$ for internal FSM port *x* – see Sections III-E, III-I, and Appendix-E.

V. Refinement 1: To Store or Not To Store Data

Parallel operations, especially quasi delay-insensitive ones, require that data be stored between sender and receiver. This Section looks at elimination of data storage to save circuit area and power while maintaining delay-insensitivity of the data exchange. As an example, we use the bidirectional communication in Fig. 5(a–b), compiled from the parallel program fragments chp{...p?x_1!y_1...} and chp{...p!y_2?x_2...}, which exchange y_1, y_2 values and store these locally in x_2, x_1.

The fat line-arrows in Fig. 5(a) mark the control flow in the compiled communication statement for the leftmost process. The elongated loops in the control flow, for instance from port r at *VAR* y_1 Joint J_{3P} to *CHAN* and back to Link L_{3P}, are indicative of *telescopic* behaviors [19] in which Links visited earlier hold data used later in the path. All the Links between data holder and user can simply transfer the data values without storing them. In the case of Fig. 5(a), *VAR* y_1's internal Link (see Fig. 2(a)) holds data for Link L_{3Q} in the peer process. Both L_{3P} and L_{3Q} store the data they receive from *CHAN*. After the data exchange by *CHAN*, each process may stop holding data for its peer, and *at its own pace* copy the received and now locally stored data into *VAR* x_1 Joint J_{4P} (leftmost process) or *VAR* x_2 (its peer). In Fig. 5(a) we store only L_{3P} and L_{3Q} data from *CHAN*, and internal Link data.

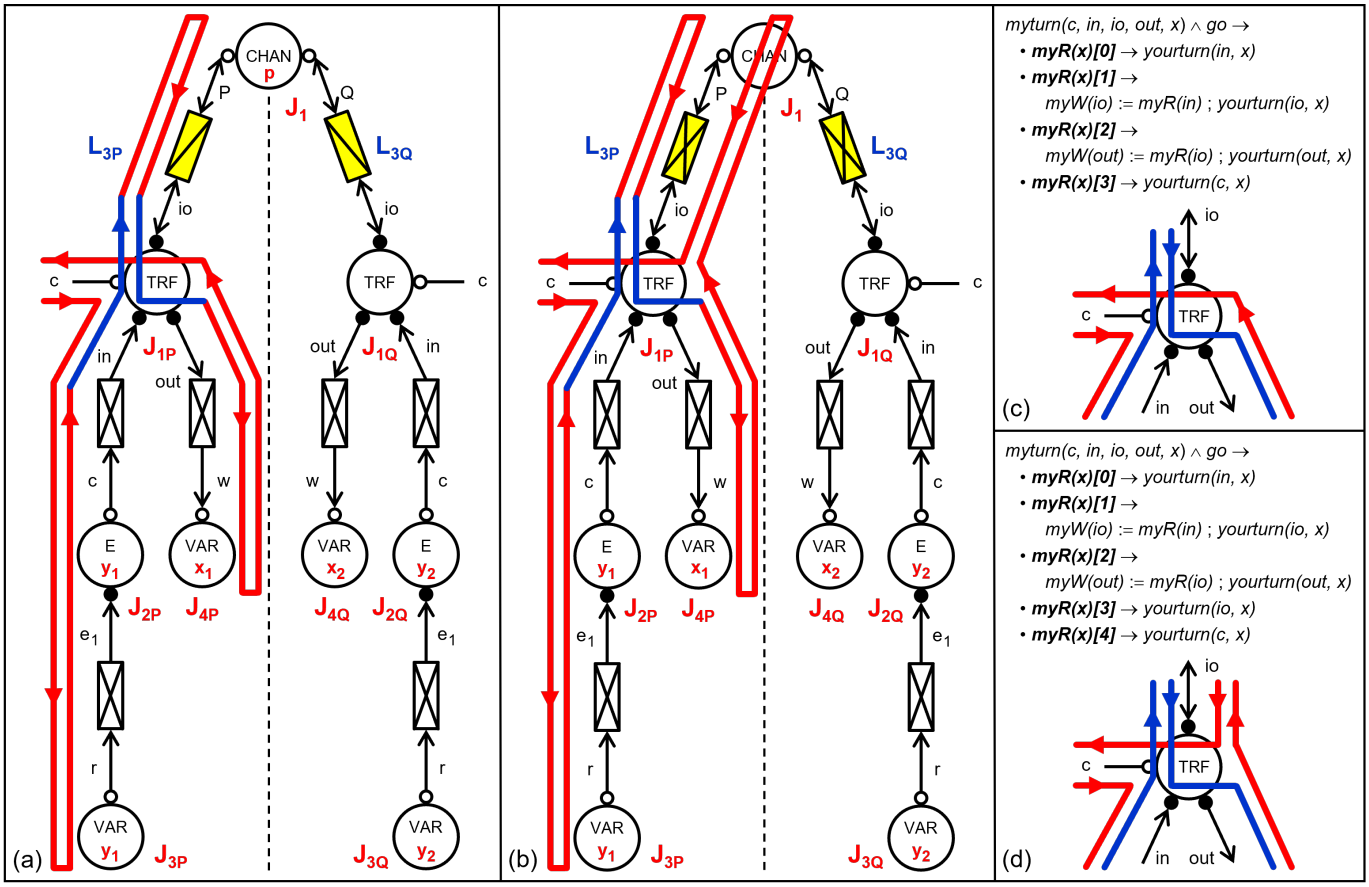

Fig. 5: Panels (a)–(b) show the same Link-Joint compilation but different Link-Joint refinements for two communicating program fragments in ACT [13], chp{...p?x_1!y_1...} (left) and chp{...p!y_2?x_2...} (right), for bidirectional channel p and local program variables x_1, x_2, y_1, y_2. The two refinements differ in Link storage and *TRF* Joints. We use rectangles with a cross for Links without data storage. Rectangles with half a cross, for L_{3P} and L_{3Q} in (a), store only data received from *CHAN*. Panel (a) uses Joint *TRF* in (c). Panel (b) uses Joint *TRF* in (d).

(a) In panel (a) we removed all data storage in the Link-Joint network, except for internal Links and data received from *CHAN* in L_{3P} and L_{3Q}. The two communicating processes follow similar execution paths. We marked the path for the process on the left with thick red and blue lines. The path starts at port c of J_{1P}, continues via J_{2P} to port r of J_{3P} to read the value of program variable y_1, which it sends via J_{2P} and J_{1P} to *CHAN* port P, where it stalls until the communication partner at port Q is ready to exchange its value of y_2 for the value of y_1. When ready, the path continues by storing the received y_2 value into the L_{3P} storage location for data from J_1. Storing the values of y_1 and y_2 in Links L_{3Q} and L_{3P} and relinquishing both Link turns altogether take one atomic *CHAN* action — and enables both partners to finish their execution paths independently, at their own pace, and without retarding each other.

(b) In panel (b) Links L_{3P} and L_{3Q} are marked as rectangles with a cross — we have now removed all but internal Link data storage. Execution for the process on the left follows the path in (a) until the *now un-stored* y_2 value is written into J_{4P} (for x_1) and execution returns to port *out* of J_{1P}. Instead of finishing, the execution revisits port P of J_1 where it stalls until the communication partner at port Q reports that it too wrote the data it received (into J_{4Q}, for x_2). When both processes report completion, execution on the left relinquishes the Link turn on L_{3P} from port P at J_1 to hand it over to port *io* at J_{1P}. Relinquishing both turns for L_{3P} and L_{3Q} is an atomic *CHAN* action. Both partners can now continue their executions independently. Execution on the left finishes at port c of J_{1P}.

In Fig. 5(b) we store only internal Link data. As a result, *VAR* y_1 must hold its data until the peer process has copied the values into *VAR* x_2 Joint J_{4Q}. The corresponding path from data holder to data user is now beyond the control flow in (a). Fig. 5(b) extends the control flow in (a) by extending Joint *TRF* and its communication protocol with *CHAN*. The elongated loops in the control flow for the leftmost process now include the path from port *r* at *VAR* y_1 to *CHAN* to port *w* of *VAR* x_1 and back to *CHAN* — to tell its peer to stop holding data.

The extension in Fig. 5(b) leads to a 4-phase communication protocol over Links L_{3P} and L_{3Q}. Because each process holds the data it sends to *CHAN* throughout the communication, we can implement each 4-phase communication as twin 2-phase communications. The persistence of data, from the moment data arrive to communication completion at each *TRF* port *c*, is reflected in the *TRF* specifications for (a) and (b) in Fig. 5(c) and (d). Note that each *TRF* is an extension of the earlier Joint *TRF* in Fig 2(b), specified in Section III-D.

There is a third solution that, like Fig. 5(a), uses 2-phase handshakes only and that, like Fig. 5(b), stores only internal Link data but that, unlike (a) and (b), constrains the network. The corresponding constraints are *delay-sensitive* and can be expressed as relative timing constraints [22] within the communication network or its surroundings, as follows.

- *static constraint:* From the moment *CHAN* exchanges data, the leftmost process and its peer must write the data they receive in *VAR* x_1 and x_2, respectively, before completing their communication parts at *TRF* port *c*.
- *dynamic constraint:* In reality, it suffices that the leftmost process writes x_1 before y_2 changes reach x_1. Though this dynamic constraint may be harder to analyze, it may point to alternative static constraints.

As a guideline for selecting one out of the three refinement solutions: Fig. 5(a) is robust and fast, (b) is robust and small, and the third solution is both fast and small, but timed.

Each solution may replace Links with data storage by Links without, and swap in different *TRF* Joints, but *none of these replacements change the topology of the compiled Link-Joint network and its syntax-directed relation to the program!*

A. Circuit Implementations for Refinement 1

Fig. 6 shows Link circuits for 2-phase communication with or without data storage. The *level-signaling bundled data* circuit implementations are done in three circuit families: Click [15], Set-Reset [20], and GasP [23]. Fig. 7(g) shows a compatible Joint circuit taking as example Joint *F* in Fig. 2(e). The top part of Fig. 7(g) implements the guards and the turn operations in the guarded command specification of *F* in Section III-G. The bottom part computes the data.

The relation between guarded command specifications and circuit implementations is as follows. Terms *myturn(p)*, *myR(p)*, *myW(p)* become circuit signals *myturn(p)*, *myR(p)*, and *myW(p)*. Term *yourturn(p)* has circuit representation *yourturn(p)*↑ ; *((myturn(p)*↓*; yourturn(p)*↓*)* , *yourturn(*p_{peer}*)*↑*)* — where p_{peer} is *A* if *p* connects to Link port *B*, otherwise *B*.

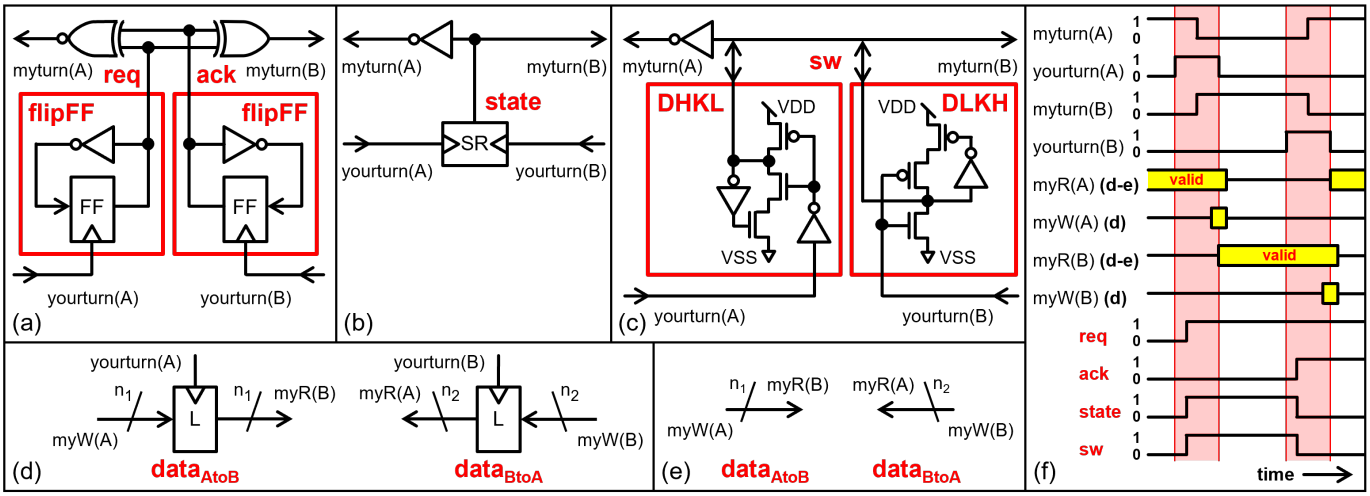

Fig. 6: Links for *2-phase level-signaling bundled-data* protocols in Click (a), Set-Reset (b), and GasP (c) with data storage (d) or without (e). The Links do double-duty for the 2- and 4-phase scenarios without data storage in Fig. 5. Data and turn pass between ports *A*, *B* by raising and lowering interface signals while maintaining the guarded command atomicity specified by the abstract Link-Joint model, e.g., { *myturn(A)=1* } *yourturn(A)*↑*; ((myturn(A)*↓ *; yourturn(A)*↓*)* , *myturn(B)*↑*)* *; yourturn(B)*↑*; ((myturn(B)*↓ *; yourturn(B)*↓*)* , *myturn(A)*↑*) etc.* Each pass, Click inverts the state of one of its *flipFF* flipflops to change *req* or *ack* and both its XOR and XNOR outputs. Set-Reset inverts the state of its *SR* latch, as does GasP using a faster SR latch split into *Drive-High-Keep-Low (DHKL)* and *Drive-Low-Keep-High (DLKH)*. Panel (d) shows the signal waveforms timed from left to right, with interface signal names in black and internal names in red. The light-red vertical bars mark the activities from rising to falling *yourturn(A)* and *yourturn(B)* signals, which act as local latch and flipflop "clocks." The yellow-colored boxes in the data waveforms indicate when data *must* be valid. The yellow boxes for *myW(A)* and *myW(B)* mark "clock" versus data setup and hold times, relevant when the Link stores the data (d). Note that *myR(A)* and *myR(B)* must be valid until the Link has stored (d) or transferred (e) the corresponding results — *myW(A)* for *myR(A)*, *myW(B)* for *myR(B)*. Crucially, and integral to Links and Joints [20], differences in how each circuit family implements this protocol are *in*-visible at the interface: same protocol, same interface!

VI. Refinement 2: Fuse or Split Joint Atomicity

Communication protocols and Joints can provide atomicity when and where needed, for as long as needed. Fig. 5 clearly demonstrates the "for as long as needed" provision: the control flow marked by the fat line-arrows in (a) and (b) shows each network operation as if it were a single atomic action — which makes sense as the network corresponds to a single communication statement in the program. This Section looks at the other end of the spectrum: an atomic Joint action, and how to split it into smaller actions. Our motivation for splitting an atomic Joint action is to facilitate its circuit implementation.

Given that each atomic action corresponds to one guarded command in the specification of the Joint, we will focus on splitting guarded commands. As an example, we use Joint *F* in Fig. 2(e), specified in Section III-G. *F* is a "heavy" Joint, with many output assignments in a single guarded command.

We can make *F* "less heavy" by splitting its assignments into separate guarded commands, while maintaining the overall command sequence. To express this requires that we refine the original Link-Joint model specified in Section III-A. We do this first, in Section VI-A. Section VI-B revisits *F* and provides a "less heavy" guarded command specification. Section VI-C and Figs. 7–8 present circuits and simulation waveforms.

A. Formalization: a Link-Joint Model for Refinement 2

Our formalization strategy is to split a guarded command into operations performed over the same Link port. This produces a set of guarded commands, each with fewer operations than the original guarded command. To maintain the original "fused" command sequence, we add a guarded command that relinquishes the turns on the Link ports, and is executed upon completion of the guarded commands in the set.

To support this strategy, we extend the original shared variable model [9] specified in Section III-A. Each Link still has two ports, *A* and *B*, and three variables, *turn*, $data_{\text{AtoB}}$, and $data_{\text{BtoA}}$ that it shares with the Joint ports at *A* and *B*. The two ports still take turns updating the Link variables, including variable *turn*. What is new is that Link variable *turn* now tracks how far along Link port *p* is in its command execution. We extend the terminology used in the guarded command specifications accordingly, as follows.

- Boolean *myturn(p)* is *true* if and only if *p* has the turn but has not yet completed its command.
- As before, *myR(p)* and *myW(p)* are data read and written by *p* — going from Link variables to *p* and vice versa.
- Boolean *midturn(p)* is *true* if and only if *p* has the turn *and* completed its command.
- Assignment *halfturn(p)* changes Link variable *turn* so that *myturn(p)* becomes *false* and *midturn(p)* becomes *true*.
- Assignment *yourturn(p)* changes *turn* so *myturn(p)* and *midturn(p)* become *false* and *myturn*(p_{peer}) becomes *true*, where p_{peer} is *A* if *p* connects to *B*, otherwise *B*.

This refinement leads to a 4-phase communication on port *p*: *myturn(p) ; halfturn(p) ; midturn(p) ; yourturn(p)*, Its peer p_{peer} sees only *myturn(p)* and *yourturn(p)* and can freely be 2-phase or 4-phase. *Joint splitting is immaterial to neighboring Joints!*

We also extend the terminology for *go*, an external Boolean for initialization, test, and debug, which is part of the guard in a guarded command and is arbitrated to permit or deny command execution. The arbitration, implicit in the original specifications, becomes visible when we split a guarded command. Because a guarded command is atomic, its execution, once started, must complete. To model atomic executions of "heavy" guarded commands with a set of multiple "less heavy" guarded commands, we add the following terms.

- Boolean *mid(go)* is *true* if and only if the arbiter has made a decision and decided to permit command execution.
- Assignment *half(go)* makes *mid(go) true*.
- Assignment *your(go)* makes *mid(go) false*.

The resulting Link-Joint model is backward compatible with the original model in Section III-A by Esimai-Roncken [9].

B. Joint F Revisited

We can now create and specify a "less heavy" version of Joint *F* in Fig. 2(e) than the version specified in Section III-G. To do this, we split the two atomic guarded commands of *F* using the strategy explained in Section VI-A, and *i*, *j* to index the *m* output Links of *F*, where $i \neq j \wedge 1 \leq i, j \leq m$.

1st guarded command — for *myR(x)[0]*

myturn(out_1..out_{m}) $\wedge$ (*go* $\vee$ *mid(go)*) $\wedge$ *myR(x)[0]*
$\wedge$ *myturn(in)* $\wedge$ (*myturn(x)* $\vee$ *midturn(x)*)
$\rightarrow$ *halfturn(in) ; half(go)*

myturn(out_1..out_{m}) $\wedge$ (*go* $\vee$ *mid(go)*) $\wedge$ *myR(x)[0]*
$\wedge$ *myturn(x)* $\wedge$ (*myturn(in)* $\vee$ *midturn(in)*)
$\rightarrow$ *halfturn(x) ; half(go)*

midturn(in, x) $\rightarrow$ *yourturn(in, x) ; your(go)*

2nd guarded command — for *myR(x)[1]*

myturn(out_i) $\wedge$ *myturn(in)* $\wedge$ (*go* $\vee$ *mid(go)*) $\wedge$ *myR(x)[1]*
$\wedge$ (*myturn(x)* $\vee$ *midturn(x)*)
$\bigwedge_{j=1..m}^{j\neq i}$ (*myturn*(out_j) $\vee$ *midturn*(out_j))
$\rightarrow$ *myW*(out_i)$:=f_i$ (*myR(in)*) ; *halfturn*(out_i) ; *half(go)*

myturn(x) $\wedge$ *myturn(in)* $\wedge$ (*go* $\vee$ *mid(go)*) $\wedge$ *myR(x)[1]*
$\bigwedge_{j=1..m}$ (*myturn*(out_j) $\vee$ *midturn*(out_j))
$\rightarrow$ *halfturn(x) ; half(go)*

midturn(out_1..out_{m}, *x*) $\rightarrow$ *yourturn*(out_1..out_{m}, *x*) ; *your(go)*

Because the *go* arbiter is released and each Link turn relinquished upon completion of *all* split commands, both the 1st and 2nd guarded command executions are atomic. When the environment stops *F* by making *go false*, it stops *F* safely — before, between, or after the 1st and 2nd guarded commands.

C. Circuit Implementations for Refinement 2

Fig. 7(h) shows a circuit for the "less heavy" (split) version of Joint *F*, specified in Section VI-B, Its ports are compatible with ports *A* and *B* of the 4-phase Link circuits in Fig. 7(a,b,c). As a comparison, Fig. 7(g) shows a circuit implementation for the "heavy" (fused) version of *F* with 2-phase ports.

Because the decision to fuse or split a Joint is irrelevant to neighboring Joints, Links may use 2-phase communication at one port, 4-phase at the other. Links (d,e,f) can connect Joint (g) ports to 2-phase port *A*, Joint (h) ports to 4-phase port *B*.

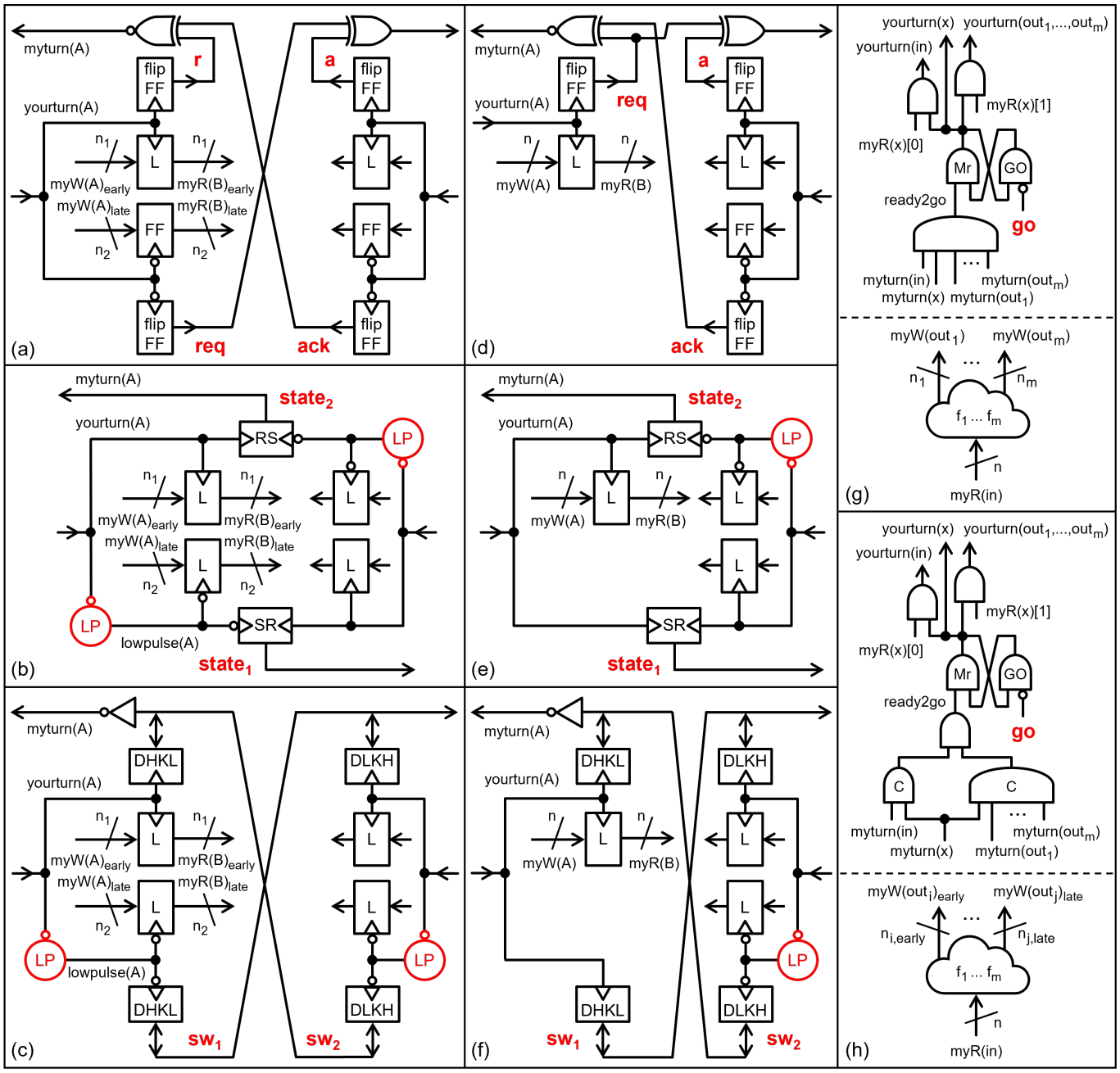

Fig. 7: Links (a,b,c) and Joint (h) use 4-phase protocols. Links (d,e,f) have 2-phase port *A*, 4-phase port *B*. Joint (g) uses 2-phase protocols. Both Joints implement Joint *F* in Fig. 2(e). The parts above the dashed line in (g,h) implement the guards and turn operations in the guarded command specifications for *F* in Sections III-G and VI-B, respectively. The parts below implement the data computations in the commands.

- Link circuits (a,b,c) use a *4-phase level-signaling bundled-data* communication protocol at both ports. Link circuits (d,e,f) each use a *2-phase level-signaling bundled-data* communication protocol at port *A* and a 4-phase version at port *B*. Both sets are implemented in Click (top), Set-Reset (middle) and GasP (bottom). Note that 4-phase communication in Set-Reset and GasP requires a low pulse generator **(LP)** — unnecessary in Click due to the use of flipflops. Replacing the Set-Reset and GasP latches for *late data* by flipflops would still require a low pulse generator, but only at the 4-phase port interface to the SR latch or the DHKL-DLKH latch pair. Differences in how each circuit family implements a given protocol stay inside the Link. If the protocol is the same, so is the interface!
- Joint circuit (g) uses a *2-phase level-signaling bundled-data* communication protocol at its external ports *in*, out_1 ... out_m, and internal port *x*. Joint circuit (h) uses a *4-phase* protocol version. Both circuits use a MrGO gate [20] to arbitrate between action denial ($\neg go$) and permission (*ready2go*). Permission, if granted, persists until internal signal *ready2go* goes low. In the 2-phase version (g), it takes one signal of $\neg myturn(in)$, $\neg myturn(x)$, $\neg myturn(out_i)$, $1 \leq i \leq m$, to lower *ready2go*. In the 4-phase version (h) it takes all $\neg myturn$ signals of ports that participate in the action, i.e., either $\{\neg myturn(in), \neg myturn(x)\}$ or $\{\neg myturn(x), \neg myturn(out_1)$ ... $\neg myturn(out_m)\}$ — as indicated by the two C element groups. Where each AND output is high when all AND inputs are high, and low when one input is low, each C output, *when low*, remains low until all inputs are high, and *when high*, remains high until all inputs are low [21]. Signal *ready2go* may be delayed to match the time for data computation, represented here as a cloud with combinational functions f_1 to f_m. As is typical for *level-signaling bundled-data*, matching half the computation delay suffices because *ready2go* is used twice per action.

Like Fig. 6, Fig. 7 uses *level-signaling bundled data* circuits, with Link implementations in Click, Set-Reset, and GasP. Earlier, in Section V-A, we showed the relation between guarded command specifications and circuit implementations for 2-phase ports. We do the same here for 4-phase ports, by relating guarded command terms defined in Section VI-A to circuit signals in Fig. 7. Terms *midturn(p)* and *halfturn(p)* have 4-phase circuit representations $\neg myturn(p) \wedge yourturn(p)$ and $yourturn(p)\uparrow ; myturn(p)\downarrow$. Term *yourturn(p)* has 4-phase circuit representation $yourturn(p)\downarrow ; myturn(p_{peer})\uparrow$ — where p_{peer} is *A* if port *p* connects to Link port *B*, and otherwise *B*. All other terms have one-to-one translations, e.g., term *myturn(p)* becomes circuit signal *myturn(p)* for 4-phase port *p*.

Splitting makes a "heavy" Joint "less heavy" in terms of its guarded command specification *and* in terms of circuit timing and analysis. Each 4-phase Link port can manage its own affairs before changing its turn signals. The C elements in 4-phase Joint *F* in Fig. 7(h) wait until all turn inputs have changed before allowing the Link ports to relinquish their turn. As a result, timing can be done by Link.[1] In contrast, the AND gates in 2-phase Joint *F* in Fig. 7(g) require additional relative timing constraints to align the Links. Fig. 8 illustrates this further. When we compare the waveforms of 4-phase port *A* in (a) to their 2-phase versions in (b) we see a robust handover of turn and valid data signals in (a), while in (b) *B* gets the turn ($myturn(B)\uparrow$) before its inputs, *myR(B)*, are valid. Splitting a "heavy" Joint facilitates its circuit implementation by modularizing its design and timing. The resulting circuits have more Link gates and more-complex Joint gates, as is obvious from Fig. 7, but the extra Link gates merely store 1-bit state information. One would expect that the "heavy" version is faster, but this may depend on delay variations, delay matching, placement, and routing.

[1]There is a catch: input data, e.g., *myR(B)* in Fig. 8(b), must be valid until computed results, e.g., $myW(B)_{\text{early}}$, have been captured by *each* Joint port. For telescope Joint *F* this is a non-issue provided input data are stored.

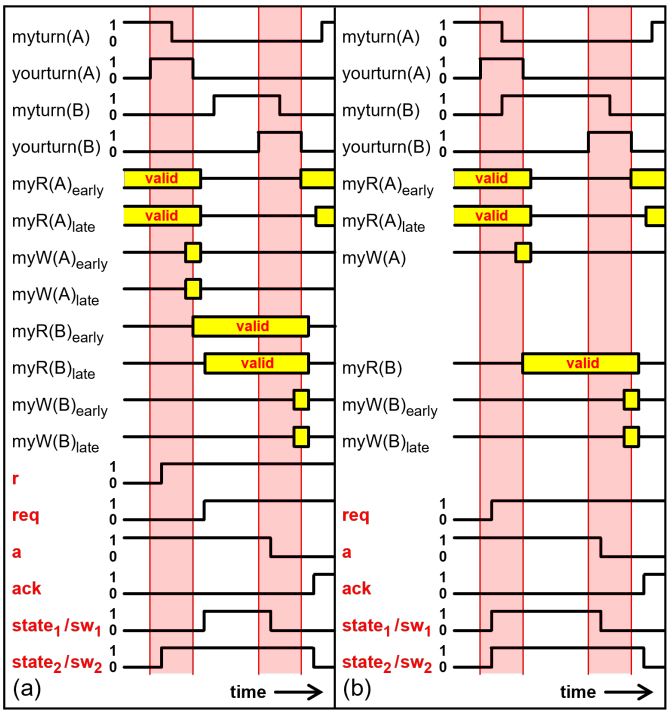

Fig. 8: Signal waveforms for Links in Fig. 7 timed from left to right. Like Fig. 6, which explains the color scheme, the waveforms start with the turn at port *A* and relinquish the turn from *A* to *B* then *B* to *A*, as indicated by the two light-red vertical bars. Waveforms in (a) are for 4-phase Links in Fig. 7(a,b,c). Those in (b) are for Links in Fig. 7(d,e,f) with 2-phase port *A*, 4-phase port *B*. Data $myR(A)_{\text{late}}$ and $myR(B)_{\text{late}}$ may become valid after $myR(A)_{\text{early}}$ and $myR(B)_{\text{early}}$, respectively — especially when flipflops store the late data.

D. Final Notes: Early and Late Data, and Testing C Elements

The circuits in Fig. 7 split 4-phase communication data into *early* and *late* data. This is a straightforward extension of the refinement in Section VI-A. Late data can supply stable data by suppressing changing data. Overlapping *early* and *late* bits may support just-in-time delay matching [10]

Scan access to state-holding C elements in Fig. 7(h) is unnecessary for the same reasons that we can avoid scan access to the many state-holding gates in superconducting Joint circuits [17]. The C elements can change temporarily during a Joint action but have the same state at each iteration of that action. This makes them *information-free* for initialization and test purposes. We can initialize the C elements and test their operations by controlling and observing Link states.

VII. Conclusion and Future Work

We are embedding Links and Joints [17]–[20] in Yale's ACT design flow [1], [13], [14] to combine asynchronous design automation with circuit flexibility. Our compiler translates hierarchical ACT programs with data-flow and control-flow into Link-Joint networks. The refinement techniques that we presented will lead to gate-level circuits with lower area and power and simpler timing. These circuits go back into ACT, for technology mapping, timing analysis, etc.

The key technical challenge in compiling ACT programs to Link-Joint networks was to find the systematic translation from handshake circuits to Links and Joints in Section III and Fig. 2. Once found, we could more or less re-use Yale's compiler. For the refinements, the key challenges were the formalization of splitting atomic actions in Section VI-A and the corresponding design of 4-phase and mixed 2-phase/4-phase circuits in Fig. 7. We will continue to formalize Link-Joint refinement levels in preparation of automated stepwise refinement and verification.

The circuit implementations in this paper use *2-phase or 4-phase, level-signaling, bundled-data* protocols in Click, Set-Reset, and GasP. Our *compile then refine* approach supports other circuit families beyond these three, and other protocols. Recent additions are synchronous and asynchronous pulse-signaling Link and Joint designs for data-flow operations in a superconducting RSFQ family [17]. These have the same ACT programs and Link-Joint networks as their CMOS versions. We are investigating Link-Joint models and CMOS circuits for *2-phase transition-signaling* and *4-phase QDI-data* protocols.

We plan eventually to implement this *compile then refine* approach as a deep embedding in ACT.

APPENDIX

Remaining specifications for Joints in this paper follow here. See Section III and Fig. 2 for context and terminology.

A. Joint REP

REP represents infinite repetition. It has startup port c and port s for the statement it repeats. Figs. 3(b) and 4(a) use *REP* as compilation for ACT-CHP command "$*$" [13]. *REP* has one guarded command that never relinquishes the turn on c.

$$myturn(c,\ s) \wedge go \rightarrow yourturn(s)$$

B. Joint PAR

PAR represents parallel composition — see Fig. 3(b). It has a basic Joint *par*, startup port c, ports s_1 to s_m for program statements it executes in parallel, and internal FSM port x to sequence its operations. *PAR* has two guarded commands.

$$myturn(c,s_1..s_m,x) \wedge go \wedge myR(x)[0] \rightarrow yourturn(s_1..s_m,x)$$
$$myturn(c,s_1..s_m,x) \wedge go \wedge myR(x)[1] \rightarrow yourturn(c,\ x)$$

C. Joint RMUX

Multiplexing Joints provide access to a shared resource, e.g., a variable or channel, from different locations in the program. *RMUX* provides read access and appears in Fig. 4. A multiplexing Joint — for read access, write access, or both — has branch ports b_1 to b_m, one per access location, a shared access port *trunk*, and internal FSM port x to sequence its two guarded commands as follows, using index i, $1 \leq i \leq m$.

$$myturn(b_i,\ trunk,\ x) \wedge go \rightarrow$$
$$myR(x)[0] \rightarrow myW(trunk) := myR(b_i)\ ;\ yourturn(trunk,\ x)$$
$$myR(x)[1] \rightarrow myW(b_i) := myR(trunk)\ ;\ yourturn(b_i,\ x)$$

D. Joint E

Joint E is for evaluating expressions in communication and assignment statements — see Fig. 3(b). It has startup and output port c, ports e_1 to e_m for connecting expression variables v_1 to v_m, and internal FSM port x to sequence two commands, specified as follows, using $g = expression(v_1..v_\mathrm{m})$.

$$myturn(c,e_1..e_\mathrm{m},x) \wedge go \wedge myR(x)[0] \rightarrow yourturn(e_1..e_\mathrm{m},x)$$
$$myturn(c,e_1..e_\mathrm{m},x) \wedge go \wedge myR(x)[1] \rightarrow$$
$$myW(c) := g\,(myR(e_1..e_\mathrm{m})\ /\ v_1..v_\mathrm{m})\ ;\ yourturn(c,x)$$

E. Joint SEL_{nondet}

SEL_{nondet} in Fig. 4 performs nondeterministic selection based on a Boolean string of one or more *true* bits arriving at port g. SEL_{nondet} has basic Joint sel_{nondet}, startup port c, ports s_1 to s_m for the program statements it may select, and internal FSM port x to sequence its operations. The built-in nondeterministic selection of guarded commands provides a simple specification, using index i, $0 \leq i < m$, which also works for deterministic selection based on a 1-hot string.

$$\begin{aligned} &myturn(c,\ g,\ s_1..s_m,\ x) \wedge go \rightarrow \\ &\quad myR(x)[0] \rightarrow yourturn(g,\ x) \\ &\quad myR(x)[1] \wedge myR(g)[i] \rightarrow yourturn(s_{i+1}),\ x) \\ &\quad myR(x)[2] \rightarrow yourturn(c,\ x) \end{aligned}$$

ACKNOWLEDGMENT

We thank Ivan Sutherland, Gary Delp, Bart McCoy, Warren Hunt, Glenn Shirley, and Quinn Morgan for feedback and comradery. We thank Rajit Manohar for early access to Yale's open-source asynchronous circuit toolkit, ACT, and for making ACT conversant with different circuit families and protocols.

REFERENCES

[1] S. Ataei, W. Hua, Y. Yang, and R. Manohar, "An open-source EDA flow for asynchronous logic," IEEE Design & Test, 38(2):27–37, 2021.

[2] A. Bardsley and D.A. Edwards, "The Balsa asynchronous circuit synthesis system," Forum on Design Languages, 2000.

[3] A. Bardsley, "Implementing Balsa handshake circuits," Doctoral thesis, University of Manchester, UK, Department of Computer Science, 2000.

[4] K. van Berkel, "Handshake circuits: an intermediary between communicating processes and VLSI," Doctoral thesis, Technical University Eindhoven, The Netherlands, 1992.

[5] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters, "Asynchronous circuits for low power: A DCC error corrector," IEEE Design & Test of Computers, 11(2):22–32, 1994.

[6] S.M. Burns and A.J. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," Advanced Research in VLSI, MIT Press, pp. 35–50, 1988.

[7] S.M. Burns, "Automated compilation of concurrent programs into self-timed circuits," Master thesis, California Institute of Technology, Computer Science Department, 1987.

[8] E.W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," Communications of the ACM, 18(8):453–457, 1975.

[9] E. Esimai and M. Roncken, "Flexible active-passive and push-pull Protocols," IEEE Embedded Systems Letters, 14(3):139–142, 2022.

[10] D. Hand, M. Moreira, H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, M. Breuer, N. Calazans, and P. Beerel, "Blade — a timing violation resilient asynchronous template," IEEE International Symposium on Asynchronous Circuits and Systems, pp. 21–28, 2015.

[11] C. A. R. Hoare, "Communicating Sequential Processes," Communications of the ACM, 21(8):666–677, 1978.

[12] J. Kessels and M. Rem, "Designing systolic, distributed buffers with bounded response time," Distributed Computing, 4:37–43, 1990.

[13] R. Manohar, "ACT hardware description language documentation," 2018, https://avlsi.csl.yale.edu/act [online].

[14] R. Manohar, "ACT tools," 2018, https://github.com/asyncvlsi/ [online].

[15] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click elements: an implementation style for data-driven compilation," IEEE International Symposium on Asynchronous Circuits and Systems, pp. 3–14, 2010.

[16] A. Peeters, "Single-rail handshake circuits," Doctoral thesis, Technical University Eindhoven, The Netherlands, 1996.

[17] M. Roncken, E. Esimai, V. Ramanathan, W. A. Hunt and I. Sutherland, "State Access for RSFQ Test and Analysis," IEEE Transactions on Applied Superconductivity, 33(5):1–7, 2023 [early access].

[18] M. Roncken and I. Sutherland, "Design and test of high-speed asynchronous circuits," Asynchronous circuit applications, J. Di and S.C. Smith, Eds. The Institute of Engineering and Technology (IET), London, UK, Chapter 7, pp. 113–171, 2020.

[19] M. Roncken, I. Sutherland, C. Chen, Y. Hei, W. Hunt Jr., C. Chau, S. Mettala Gilla, H. Park, X. Song, A. He, and H. Chen, "How to think about self-timed systems," Asilomar Conference on Signals, Systems, and Computers, pp. 1597–1604, 2017.

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

[21] J. Sparsø, "Introduction to asynchronous circuit design," DTU Compute, Technical University of Denmark, 2020 [available online from DTU].

[22] K. Stevens, R. Ginosar, and S. Rotem, "Relative timing," IEEE Transactions on Very Large Scale Integration Systems, 11(1):129–140, 2003.

[23] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control." IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 46–53, 2001.