How to Think about Self-Timed Systems

Marly Roncken*, Ivan Sutherland*, Chris Chen*, Yong Hei*†, Warren Hunt Jr.‡, and Cuong Chau‡,
with Swetha Mettala Gilla*, Hoon Park*, Xiaoyu Song*, Anping He§, and Hong Chen¶

*Portland State University, Portland, Oregon, USA
†Institute of Microelectronics Chinese Academy of Sciences, Beijing, China
‡The University of Texas at Austin, Austin, Texas, USA
§School of Information Science & Engineering, Lanzhou University, Lanzhou, China
¶Institute of Microelectronics, Tsinghua University, Beijing, China

***Abstract*—Self-timed systems divide nicely into two kinds of components: communication links that transport and store data, and computation joints that apply logic to data. We treat these two types of self-timed components as equally important. Putting communication on a par with computation acknowledges the increasing cost of data transport and storage in terms of energy, time, and area. Our clean separation of data transport and storage from logic simplifies the design and test of self-timed systems. The separation also helps one to grasp how self-timed systems work. We offer this paper in the hope that better understanding of self-timed systems will engage the minds of compiler, formal verification, and test experts.**

I Introduction

On-chip communication over distance has inherent delay. Self-timed systems accommodate this delay by sending validity signals alongside data to bind or release the data. We package data transport wires and their associated validity signals into a communication component we call a *link*. Whereas a self-timed component for computation tends to be compact, a link often spans a long distance. Nevertheless, like all self-timed components, a link reports completion of its tasks. For instance, when "filled" with data at its input a link reports completion of this fill task at both its input and output ends.

Except for its delay, a link's communication behavior is entirely independent of its length. Albeit spread over space, we treat links as first class components, just like components for computation. Design and test gain simplicity by elevating communication to a status equal to that of computation [6].

We will first describe links, and then focus on how to specify and implement many types of components for computation, including flow control. We call a component for computation a *joint*, because joints are the meeting points for links to coordinate states and exchange data.

This paper intends to give readers (1) sufficient detail to model their own self-timed systems in terms of links and joints, and (2) easy access to the benefits of self-timing. We carefully refrain from specifying one of the many families of self-timed handshake communication circuits because their implementation differences can remain "under the hood" of a link and joint model [6].

II The Role of Links

A link transports data and their associated validity signals. The validity signals are stored in the link as a FULL or EMPTY state. A FULL state indicates that the link's data signals are stable and valid. The link presents a FULL state indicator at its output end to let a receiver know whether or not it is FULL. An EMPTY state indicates that the link's data have been released and may be replaced by fresh data. The link presents an EMPTY state indicator at its input end to let a sender know whether or not it is EMPTY. Our links are one-to-one connections, with data flowing always in the same direction. Links may, but need not, store the data they transport.

Figure 1 draws each link with a rectangle, to remind us that it stores state and possibly also data. The rectangle is long to remind us of the link's transport delay. Colored rectangles represent FULL links, and white rectangles represent EMPTY links. A link stores data unless its rectangle is crossed-out as in Figure 2(f). Figure 1 draws joints as stick figures, each of which coordinates the actions of two links. The arrows indicate the direction in which the data flow. The link-joint configurations in Figure 2 represent joints as circles.

A link has two actions. A "fill" event with data at its input causes the link to store the data and become FULL. The link's EMPTY state indicator, at the input end, is de-asserted immediately. Because of transport delay, the link's FULL state indicator, at its output end, is asserted some time later. Likewise, a "drain" event at its output causes the link to become EMPTY. When drained, the link's FULL state indicator, at the output end, is de-asserted immediately. Because of transport delay, the link's EMPTY state indicator, at its input end, is asserted some time later.

Figure 1 shows the action of links coupled by a simple joint that forms links into a first-in-first-out buffer configuration, or FIFO. Data in such a FIFO reside in the links that are FULL. Any subset of the links may be FULL at any one time, and so such a FIFO is elastic. Elastic FIFOs decouple the timing of arriving data from the timing of departing data.

Links serve admirably for transport as well as temporary state and data storage.

978-1-5386-1823-3/17/$31.00 ©2017 IEEE

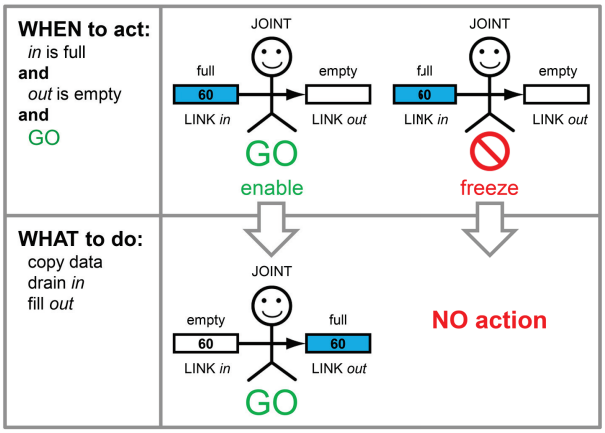

Figure 1 This picture illustrates the action of a simple joint between two links. The stick figure represents the joint, and the rectangles represent the links. FULL links are colored and EMPTY links are white. This joint can act only when its input link is FULL and its output link is EMPTY, as shown in the upper panel. When it acts, it copies data from its input link to its output link, as illustrated in the left column. The action disables itself. A joint whose *go* signal (GO) is de-asserted never acts, as illustrated in the right column.

III Joints

A joint can have many input links and many output links as illustrated in Figure 2(a). A joint may act when some or all of its input links are FULL and some or all of its output links are EMPTY. When it acts, it computes results from data it gets from its FULL input links, and passes these results to some or all of its EMPTY output links. It then fills selected EMPTY output links, and drains selected FULL input links, destroying the conditions that enabled this action. Links that it fills or drains begin their data transport actions to carry results away or fetch fresh input data.

We describe each action of a joint with a loose guarded command formalism. Because joints can be seen as processes that communicate through links, we anticipate that systems with links and joints blend Hoare's communicating sequential processes (CSP) [3] with Dijkstra's guarded commands [2]. Here is our guarded command specification for the single action of the joint in Figure 1:

action:
  when
    FULL(*in*)
    and EMPTY(*out*)
  do
    fill(*out* with data(*in*))
    and drain(*in*)

Because all actions have *go* signals, its *go* signal is an implicit part of every guard. Some joints have more than one action, each with its own guard. A joint executes its actions one at a time, each time making an arbitrary selection between actions with a valid guard. Sections III-A to III-E discuss a variety of joints. Aside from arbitration, a joint is storage-free.

III-A Broadcast Joint

The simplest joint acts when *all* its input links are FULL, providing the joint with data, and *all* its output links are EMPTY, providing it with space for computed results. When such a joint acts, it fills *each* of its output links with a result computed for that link, and it drains *all* its input links. In other words: all its links participate in the action. We call such a joint a "broadcast joint."

A broadcast joint with one input and one output link can form an elastic first-in-first-out buffer, or FIFO, as shown in Figure 1. A FIFO copies data from its input link to its output link. Given that each of its links can store at most one data item, the FIFO in Figure 1 can store up to two separate data items. The FIFO's outgoing data stream retains the data and sequence of its incoming data stream. By connecting n such FIFOs in series, using the output link of each FIFO as the input link to the next FIFO, one can make a longer FIFO that can store between zero and $n+1$ data items. This elasticity in storage capacity decouples input timing from output timing, and can be used to bridge different timing domains.

A broadcast joint with one output link and several input links might serve to combine the incoming data arithmetically, for example by adding them, as illustrated in Figure 2(b-top).

A broadcast joint with one input and several output links can serve to separate parts of a data structure, as suggested by the example in Figure 2(b-bottom). This example might be used in an instruction decoder to separate each incoming instruction into parts: opcode, index values, and address.

At the beginning of Section III, we gave a simple guarded command specification for the joint in the FIFO of Figure 1. This simple specification is silent about the structure of the data. The data structure is irrelevant for the operation as a FIFO, but is relevant when the FIFO is implemented in silicon or type-checked for compatibility as a sub-part of a system. In this paper, we include data structures and type information only when they promote understanding. The guarded command specification contains an indirect reference to the joint's computation or "copy" function, f, using the fact that $f(\mathrm{data}(in)) = \mathrm{data}(in)$. Below, we give an example of a specification with an explicit computation function, *add*, for the joint in Figure 2(b-top):

define $add(n_1,n_2) = n_1 + n_2$
action:
  when
    FULL($number_1$)
    and FULL($number_2$)
    and EMPTY(*sum*)
  do
    fill(*sum* with *add*(data($number_1$),data($number_2$))
    and drain($number_1$)
    and drain($number_2$)

A corresponding circuit follows in Figure 3(a).

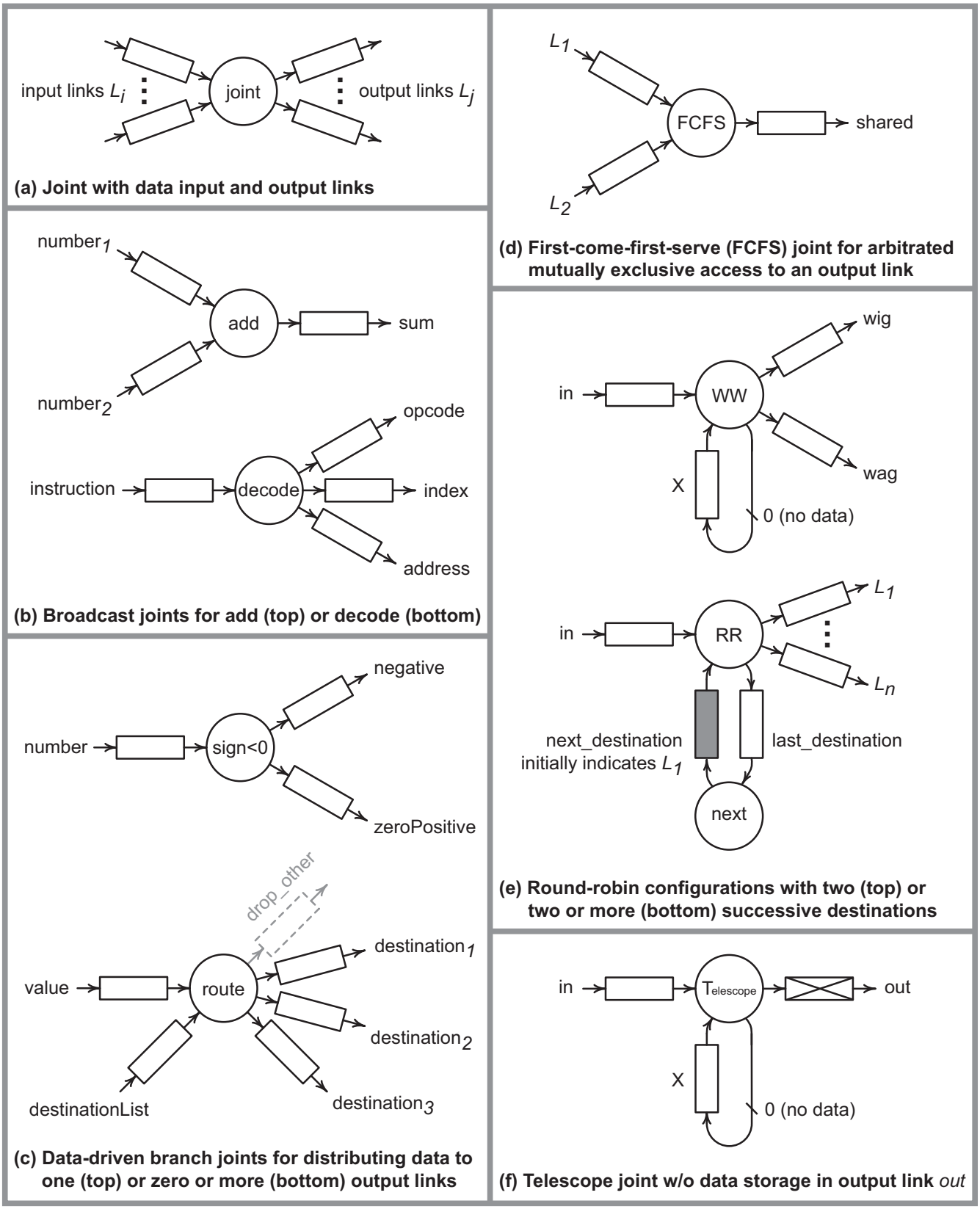

Figure 2 Configurations with links for communication and storage and joints for computation and flow control.

III-B Data-Driven Branch

A joint with one or more input links and several output links can serve as a data-driven branch by selecting different output links according to data input values. For example, the joint in Figure 2(c-top) sends negative input values to one output link and zero or positive input values to another output link. Figure 2(c-bottom) shows an example where one input link, *value*, contains the data for the selected output links, and the other input link, *destinationList*, contains the selection information. This joint passes the data to all of its selected output links, discarding mis-addressed selections.

We can design these joints with one or more mutually exclusive actions. In a design with *one* action, the joint acts when *all* its input links are FULL and *all* its output links are EMPTY. When it acts, it fills *only* selected output links with the results computed for them, and it drains *all* its input links. The joint in Figure 2(c-top), designed with *one* action, can be specified by the following guarded command:

action:
  when
    FULL(*number*)
    and EMPTY(*negative*)
    and EMPTY(*zeroPositive*)
  do
    if (*sign* < 0) fill(*negative* with data(*number*))
    else fill(*zeroPositive* with data(*number*))
    fi
    and drain(*number*)

A corresponding circuit follows in Figure 3(b-top). Another design, with two actions, follows in Figure 3(b-bottom). The specification of this design has *two* guarded commands — one per action. The guards need look only whether or not the *selected* output link is EMPTY:

define *sign_is_negative* $=$ (*sign* < 0)
action-1:
  when
    FULL(*number*)
    and EMPTY(*zeroPositive*)
    and not(*sign_is_negative*)
  do
    fill(*zeroPositive* with data(*number*))
    and drain(*number*)
action-2:
  when
    FULL(*number*)
    and EMPTY(*negative*)
    and *sign_is_negative*
  do
    fill(*negative* with data(*number*))
    and drain(*number*)

Advantages and disadvantages of these two specifications and their circuit implementations follow in Section IV.

III-C First-Come-First-Serve Arbitrated Merge

Some system operations require mutually exclusive access to a shared resource. Self-timed systems use a special circuit, known as a *mutual exclusion circuit* or *arbiter*, which grants mutually exclusive access on a first-come-first-serve basis [7]. Joint FCFS in Figure 2(d) contains such an arbiter to provide its two input links, L_1 and L_2, with mutually exclusive access to its output link, *shared*. We may use the arbiter to decide which FULL input link to grant when data *as well as* space are available, i.e. when the output link is EMPTY, or to decide *regardless* of the FULL or EMPTY state of the output link.

We specify joint FCFS, with the arbiter applied when data as well as space are available, as two guarded commands whose guards may be valid simultaneously. The specification formalism for guarded commands arbitrates on the basis of valid guards. Because its arbitration mechanism is already present in the formalism, this FCFS specification can be adopted easily in a formal model and verification framework [1].

define
  data_and_select $=$
    (data(L_1),*select*$_1$,data(L_2),*select*$_2$)
action-1:
  when
    FULL(L_1) **and** EMPTY(*shared*)
  do
    fill(*shared* with *data_and_select*)
    and drain(L_1)
action-2:
  when
    FULL(L_2) **and** EMPTY(*shared*)
  do
    fill(*shared* with *data_and_select*)
    and drain(L_2)

Specifying an FCFS joint that applies arbitration directly between the input links requires an explicit arbiter definition:

define
  (*grant*$_1$,*grant*$_2$) $=$
    arbiter(FULL(L_1),FULL(L_2),*grant*$_1$,*grant*$_2$)
  data_and_select $=$
    (data(L_1),*select*$_1$,data(L_2),*select*$_2$)
action-1:
  when
    grant$_1$ **and** EMPTY(*shared*)
  do
    fill(*shared* with *data_and_select*)
    and drain(L_1)
action-2:
  when
    grant$_2$ **and** EMPTY(*shared*)
  do
    fill(*shared* with *data_and_select*)
    and drain(L_2)

This last specification omits an arbiter definition, but hints at its presence. A circuit implementation for joint FCFS with arbitration directly on the input links follows in Figure 3(c) and is discussed in Section IV.

III-D Alternating Branch

Instead of using data to drive the flow of data, as discussed in Section III-B, one can use control. For example, "wig-wag" joint, WW, in Figure 2(e-top) sends data from its input link, *in*, alternately to two separate output links, called *wig* and *wag*.

Internal link *X* remembers whose turn is next. WW uses *wig* when *X* is EMPTY or *wag* when *X* is FULL. Because *X* is an input link of the joint, WW could receive data over *X* and can access the FULL state indicator at the receiver's end of *X*. Likewise, because *X* is an output link of the joint, WW could send data over *X* and can access the EMPTY state indicator at the sender's end of *X*. In this case, the link's EMPTY or FULL state suffices to remember whether to *wig* or *wag*. Therefore, no additional data are sent or received over *X*, as indicated by the "0-width" notation in Figure 2(e-top).

Joint WW can be designed with two mutually exclusive actions, and specified by the following two guarded commands:

action-1:
  when
    FULL(*in*)
    and EMPTY(*X*)
    and EMPTY(*wig*)
  do
    fill(*wig* with data(*in*))
    and fill(*X* without data)
    and drain(*in*)
action-2:
  when
    FULL(*in*)
    and FULL(*X*)
    and EMPTY(*wag*)
  do
    fill(*wag* with data(*in*))
    and drain(*in*)
    and drain(*X*)

A corresponding circuit follows in Figure 3(d).

The link-joint configuration in Figure 2(e-bottom) generalizes Figure 2(e-top) by sending data in turn from its input link, *in*, to n separate output links, L_1 to L_n, for $n \geq 2$. It replaces link X of Figure 2(e-top) by a link-joint-link triple, with links *last_destination* and *next_destination*, and broadcast joint *next*. Given a destination indicator for the last turn, joint *next* computes a destination indicator for the next turn. Both indicators are treated as data. Figure 2(e-bottom) replaces the control-driven wig-wag joint, WW, of Figure 2(e-top) by a data-driven "round-robin" joint, RR.

Figure 2(e-top) initializes internal link X as EMPTY, making *wig* the first destination. One can initialize X as FULL to make *wag* the first destination. Likewise, Figure 2(e-bottom) initializes internal links *last_destination* and *next_destination* as EMPTY and FULL, with data indicating L_1 as first destination. One can initialize the data for another first destination.

Wig-wag and round-robin configurations can improve throughput. Take, for instance, a joint with a complex computation function that takes too long to finish, say more than one but less than two times the target action time. Such a joint would jeopardize the throughput of the system. But we could use two such joints, and use a wig-wag branch to send data alternately to each, and a wig-wag merge to receive the alternately computed results. The alternate service of the wig-wag configuration gives each joint twice as much time to finish its computation, and maintains system throughput.

III-E Telescope Joint

One may wish to perform certain data operations in sequence without storing intermediate results. This can be accomplished using "telescope" joints.[1] For example, joint *Telescope* in Figure 2(f) sends data from its FULL input link, *in*, to its EMPTY output link, *out*. But it avoids draining *in*, until *out* has been filled and drained again, thus keeping the data sent by *in* and the data received by *out* valid and stable. Internal link *X* remembers whether or not the current data have been sent. Because link *in* guarantees the data's validity, link *out* can avoid storing the data, as indicated by the crossed-out rectangle in Figure 2(f). Link *out* still stores its FULL and EMPTY state information, which is why Figure 2(f) still draws link *out* with a rectangle. Joint *Telescope* has two actions specified as follows:

action-1:
  when
    FULL(*in*)
    and EMPTY(*X*)
    and EMPTY(*out*)
  do
    fill(*out* with data(*in*) without storage)
    and fill(*X* without data)
action-2:
  when
    FULL(*in*)
    and FULL(*X*)
    and EMPTY(*out*)
  do
    drain(*in*)
    and drain(*X*)

[1]The term "telescope" was introduced in the PhD thesis of Swetha Mettala Gilla [4]. The name refers to the behavior of a pipeline with telescope joints. The forward extension by FULL links and the reverse shortening by EMPTY links are reminiscent of the extension (unfolding) and shortening (folding) of a jointed telescope with sliding tubes.

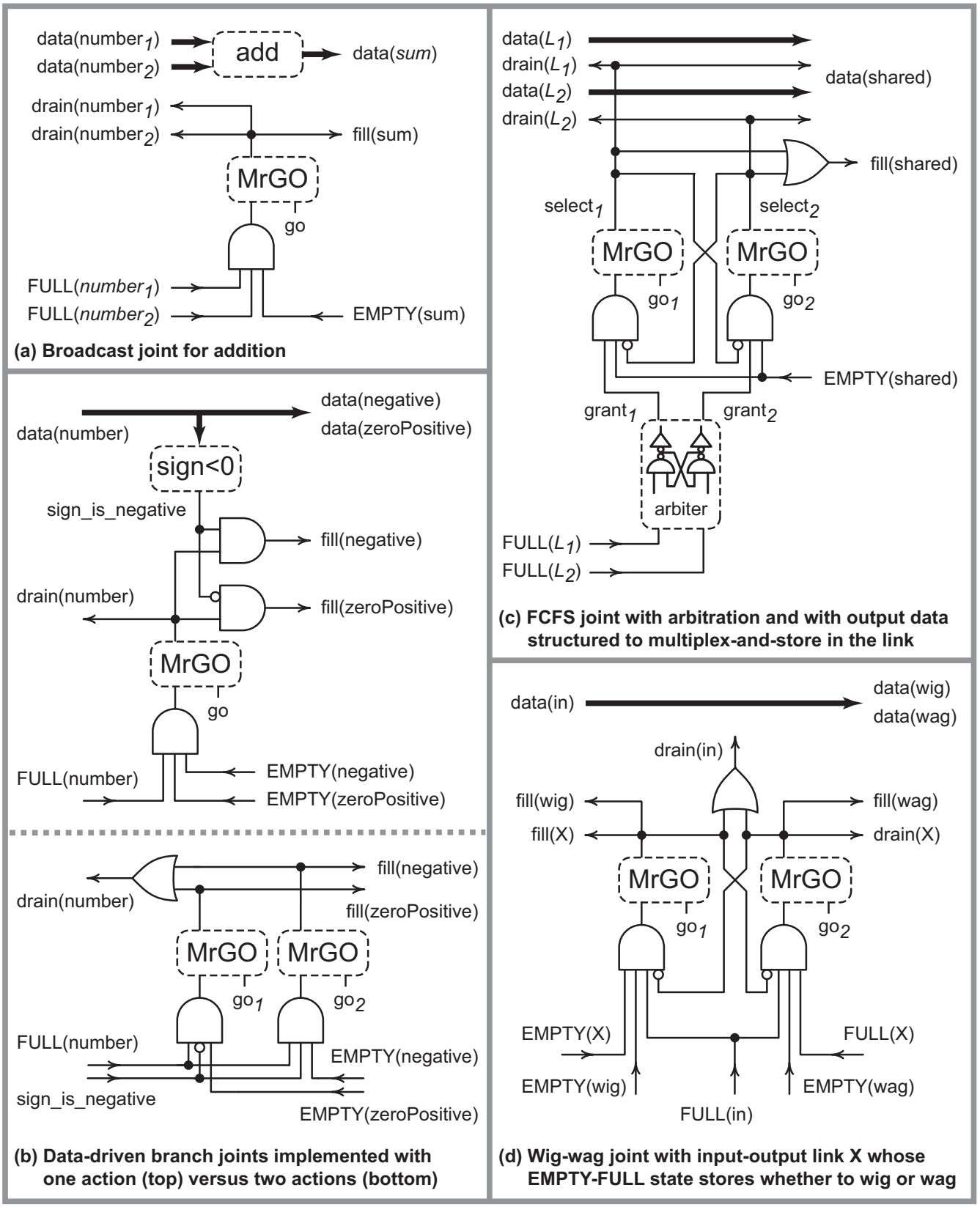

Figure 3 Gate-level circuit outlines of joints for various combinational logic computations and flow control operations.

IV Under the Hood

Figure 3 shows circuit implementations for the guarded command specifications of joints discussed in Section III. We include circuit implementations in this paper to indicate how our assumptions about mutually exclusive actions at the dataflow and link and joint specification levels translate to the lower circuit level. The gate-level circuits are drawn so as to emphasize their relation to the corresponding guarded command specifications. Sections IV-A to IV-D discuss them in the order of Figure 3. Section IV-E discusses circuit-level timing. The most important property of our circuit-level timing assumptions is that they can be kept "under the hood."

IV-A Broadcast Joint for Addition

The gate-level circuit in Figure 3(a) implements the broadcast joint for addition shown in Figure 2(b-top) and specified by the guarded command given at the end of Section III-A.

As indicated in Figure 1, each guarded command, or action, has a unique *go* signal AND-ed with its guard. The circuit in Figure 3(a) shows the guard as three guard terms entering a 3-input AND gate, followed by a MrGO circuit AND-ing in the *go* signal. More details about MrGO follow in Section IV-F.

IV-B Data-Driven Branch with One Versus Two Actions

The two gate-level circuits in Figure 3(b) implement the data-driven branch shown in Figure 2(c-top) and specified in Section III-B. Figure 3(b-top) implements the *one* and only action of the first guarded command specification in Section III-B. Figure 3(b-bottom) implements the *two* actions of the second guarded command specification in Section III-B. As in the broadcast joint, the circuits show each guard as a wide AND gate, combining the guard's terms, followed by a MrGO circuit AND-ing in the *go* signal. The guards of separate actions have separate MrGO circuits and separate *go* signals.

Note that the computation for selection signal *sign_is_negative* has more time to stabilize in Figure 3(b-top) than it has in Figure 3(b-bottom). This is an advantage for the version with *one* action. On the other hand, the *one*-action version has the disadvantage of waiting for both output links to be EMPTY, whereas the version with *two* actions waits for EMPTY on only the selected output link. In Figure 3(b-bottom), the separate *go* signals can stop executions for one branch and continue executions for the other branch, which may be valuable for test and debug.

IV-C Arbitrated Merge with Structured Data

The gate-level circuit in Figure 3(c) implements the FCFS joint in Figure 2(d). More specifically, it implements the guarded command specification in Section III-C with arbitration directly between the input links. For most system designs, this implementation is fair to greedy input links — an advantage it has over alternative designs that arbitrate between the guards. As in the previous joint implementations, each guard has a wide AND gate, combining the guard's terms, followed by a MrGO circuit AND-ing in its *go* signal.

Unlike the previous joint implementations, the guards in Figure 3(c) are cross-coupled: the negation of each guard is part of the other guard. The cross-coupled guards give the output link time to start its fill action, de-assert its EMPTY state indicator, and disable the other guard *before* the granted action ends — thus preserving its mutually exclusive access.

As specified in Section III-C, the data sent to the output link, data(*shared*), are structured so that the link itself can multiplex and store the selected data. This FCFS joint forwards the data from both inputs, data(L_1) and data(L_2), along with the selection signals, represented by $select_1$ and $select_2$.

IV-D Alternating Branch with Input-Output Link

Figure 3(d) implements the control-driven wig-wag joint, WW, in Figure 2(e-top), as specified in Section III-D. Its organization is similar to that of the *two*-action data-driven branch circuit in Figure 3(b-bottom), with additional cross-coupling of the guards, as in the arbitrated merge circuit of Figure 3(c).

Cross-coupled guards generally give links that are filled or drained by a selected action extra time to disable the joint's other actions before the selected action ends, thus preserving mutually exclusive executions of a joint's actions. As illustrated by Figure 3(b-bottom), some joint implementation with multiple actions omit cross-coupled guards. Cross-coupled guards are required only when *partial* completion of a joint's action can enable another action in the joint — e.g.:

- The data-driven branch circuit in Figure 3(b-bottom) omits cross-coupled guards, because each of its partially complete actions either de-asserts the EMPTY state indicator of the selected output link, leaving the other action as disabled as before, or de-asserts the FULL state indicator of the input link, disabling both actions.
- The wig-wag circuit in Figure 3(d) requires cross-coupled guards. A partially complete action-1, starting with link *in* FULL and the other links EMPTY, could fill link *X* and generate a temporary state with links *in* and *X* FULL and links *wig* and *wag* EMPTY, thus enabling action-2 — *if the cross-coupled guards were absent*.

The need for cross-coupled guards can be determined syntactically from a joint's guarded command specification.

IV-E Circuit-Level Timing

In Sections IV-C and IV-D, we explained that the cross-coupled guards in the circuit implementations in Figures 3(c) and 3(d) give links *time* to settle into their new FULL or EMPTY states before the current joint action ends and another starts. One might ask: how much time?

To answer that question requires a maximum path delay analysis for filling and draining the near-ends of the links involved

in the selected joint action and for propagating changes in the FULL or EMPTY state indicators up to a point where these disable another action in the joint. The cross-coupled guards *must give* a settle time greater than the analyzed maximum path delay. To compute the settle time *given* by each cross-coupled guard, we use a minimum path delay analysis for filling and draining the near-ends of the links involved and for propagating changes in the FULL or EMPTY state indicators up to the point where these release the cross-coupled guard to enable other actions in the joint. We use static timing analysis and repair procedures that insert delay when and where needed to ensure that the *given* time is longer than the *must give* time.

Similar questions and answers about circuit-level timing apply to other parts of the circuit implementations in Figure 3. Using model checking and formal verification techniques, one can identify the timing assumptions on which these circuit implementations depend, and prove their soundness and completeness — see for instance [5] and its citations. Most importantly, it is possible to keep circuit-level timing assumptions "under the hood" and avoid elevating them to the dataflow and link-joint specification and verification framework.

IV-F MrGO

MrGO, pronounced "Mister GO," makes it possible to separate initialization from execution. When its *go* input is de-asserted, MrGO disables its associated action. When all actions are disabled, we are free to change the state of any link to initialize it. When initialization is complete, asserting the *go* signal to MrGO returns control of the action to its guard. Because the system is self-timed, we can assert *go* signals in any sequence, or concurrently in groups.

Stopping a running self-timed system requires arbitration. For that purpose, MrGO has an arbiter. The arbiter in MrGO decides whether to continue or stop the action. The cross-coupled NAND gates in the arbiter representation in Figure 3(c) are a reminder that an arbiter has state. Because it has state, arbiter inputs can hog an arbiter.

Because arbiter inputs might hog the arbiter, and because we might want to change the value of an arbiter input for the purpose of initialization or test, the position of MrGO matters. We position MrGO after the guard. Because each action disables its guard, relinquishing control of MrGO, this position guarantees that a disabled *go* signal can take control of MrGO after at most one action. Thus, after at most an arbitration delay and one action delay, MrGO will disable its action and prevent it from changing the state of its links. Disabling all actions associated with a link allows us to set the state of the link without interference from its joints.

Thus far, only a fraction of our tests involve stopping a *running* self-timed system. Most tests end with the system inactive. When the system is inactive, a disabled *go* signal can take control of MrGO immediately. For test details, see [6], [4].

V Conclusion

This paper intends to make the link and joint model of self-timed circuits and systems accessible to all, including those already familiar with designing self-timed systems in specific circuit families [8]. Our aspirations are:

- that the software community will use links and joints as design and compilation targets for concurrent algorithms;
- that the test community will leverage each action's *go* control to develop algorithms for system-level testing;
- that the formal verification community will support such design and test capabilities with a unified specification and verification framework;
- and finally, because links accommodate any transport delay imposed by their length, that the layout community will place and route systems for average speed and power.

Automatic compilation of hardware systems distributed over space requires relief from the tyranny of the clock [9]. Self-timing provides such relief by eliminating timing artifacts endemic in clocked design. Artifacts such as timing closure and multiple clock domains result from the clocked design paradigm's false assumption of simultaneity over space. In contrast, self-timing avoids these artifacts by embracing the fundamental truth that time and space are intimately related.

Acknowledgment

This research was funded in part by DARPA, "Flexible Specification, Analysis, and Implementation of Self-Timed Circuits," sponsor award UTA17-000001, and in part by the Portland State University Foundation.

References

[1] Cuong Chau, Warren Hunt Jr., Marly Roncken, and Ivan Sutherland. A Framework for Asynchronous Circuit Modeling and Verification in ACL2. In O. Strichman and R. Tzoref-Brill, editors, *Haifa Verification Conference (HVC)*, LNCS 10629, pages 3–18. Springer International Publishing, 2017.

[2] Edsger Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[4] Swetha Mettala Gilla. *Silicon Compilation and Test for Dataflow Implementations in GasP and Click*. PhD thesis, Electrical and Computer Engineering, Portland State University, Defended 3 November, 2017, unpublished.

[5] Hoon Park, Anping He, Marly Roncken, Xiaoyu Song, and Ivan Sutherland. Modular Timing Constraints for Delay-Insensitive Systems. *Journal of Computer Science and Technology*, 31(1):77–106, January 2016.

[6] Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland. Naturalized Communication and Testing. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 77–84, 2015.

[7] Charles Seitz. Chapter 7: System Timing. In C. Mead and L. Conway,, *Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

[8] Jens Sparsø and Steve Furber (Eds.). *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[9] Ivan Sutherland. The Tyranny of the Clock. *Communications of the ACM*, 55(10):35–36, October 2012.