2015

Modular Timing Constraints for Delay-Insensitive Systems

Hoon Park
Portland State University, parkhoon@gmail.com

Anping He
Lanzhou University

Marly Roncken
Portland State University

Xiaoyu Song
Portland State University

Ivan Sutherland
Portland State University

Citation Details

Park, Hoon; He, Anping; Roncken, Marly; Song, Xiaoyu; and Sutherland, Ivan, "Modular Timing Constraints for Delay-Insensitive Systems" (2015). *Electrical and Computer Engineering Faculty Publications and Presentations*. 306.
https://pdxscholar.library.pdx.edu/ece_fac/306

Modular Timing Constraints for Delay-Insensitive Systems

Hoon Park[1,2], Anping He[3], Marly Roncken[1], Xiaoyu Song[2], and Ivan Sutherland[1]

[1]Asynchronous Research Center, Portland State University, 97201, U.S.A.
[2]Department of Electrical & Computer Engineering, Portland State University, 97201, U.S.A.
[3]School of Information Science & Engineering, Lanzhou University, 730000, China

E-mail: parkhoon@gmail.com, heap@lzu.edu.cn, marly.roncken@gmail.com, {song@ece, ivans@cecs}.pdx.edu

Abstract—**This paper introduces ARCtimer, a framework for modeling, generating, verifying, and enforcing timing constraints for individual self-timed handshake components. The constraints guarantee that the component's gate-level circuit implementation obeys the component's handshake protocol specification. Because the handshake protocols are delay insensitive, self-timed systems built using ARCtimer-verified components are also delay insensitive. By carefully considering time locally, we can ignore time globally. ARCtimer comes early in the design process as part of building a library of verified components for later system use. The library also stores static timing analysis (STA) code to validate and enforce the component's constraints in any self-timed system built using the library. The library descriptions of a handshake component's circuit, protocol, timing constraints, and STA code are robust to circuit modifications applied later in the design process by technology mapping or layout tools. In addition to presenting new work and discussing related work, this paper identifies critical choices and explains what modular timing verification entails and how it works.**

Keywords—**self-timed circuits, delay-insensitive systems, model checking, timing analysis, design patterns**

I Introduction

Nearly all modern digital computers march to the beat of a "clock." The computer clock divides each second into a few billion "clock periods" just as a school bell divides each day into fixed-length class periods. A fifty-five minute class period is so useful for scheduling students and classrooms that educators rarely ask if it is best for learning. In reality, fifty-five minutes is either too short or too long.

We are one of a few research groups who study how to replace the rigid clock with more flexible "self-timed" regimes. Self-timed systems allow each small task to take its own natural time just as "self-paced" learning allows each student to learn at his or her own pace. Easy tasks finish quickly and take little energy. Difficult tasks require more time and energy.

We design our self-timed systems using circuit components connected through local communication channels. The components use handshake protocols to coordinate their activities and to exchange data through the communication channels. The "self-paced" operations of the system are delay insensitive provided the components follow the handshake protocols.

We partition the verification of such a system into two parts:

- **a higher-level system part**, at the protocol level, to verify that the network of handshake components and their protocols meet the requirements of the system, and
- **a lower-level component part**, at the circuit level, to verify that the network of logic gates and wires and their delays meet the component's protocol description.

This paper describes how we do the lower-level verification in advance of system design to build a library of verified components for later system use.

The higher-level system verification part applies to digital circuits broadly. A general-purpose analysis system, such as the ACL2 modeling and theorem proving system[1], can model and verify this part in terms of cooperating finite state machines, as is done in the formal verification of microprocessors [1]. This approach is scalable to very large systems, as shown on contemporary x86 systems [2]. The key message in the context of this paper is that the lower-level component verification allows the higher-level system verification part to ignore all circuit and timing information. By carefully considering time locally we can ignore time globally.

We further partition the lower-level component verification part into three sub-parts, the last of which is the main focus of this paper. The first sub-part verifies transistor-level implementations against their gate-level descriptions. The second sub-part verifies analog behavior as logical signal transitions. The third sub-part verifies the gate-level logical signal transitions against the component's handshake protocol description.

The first sub-part, verifying at the transistor level, is quite general. For this sub-part, we can re-use existing methods in logic verification for synchronous datapath and control circuits, like [2]. The second sub-part, verifying analog behavior, is addressed in [3]. This sub-part is of lesser importance for our self-timed circuits, because we design our circuits using the theory of Logical Effort [4]. As a result, our circuits come with an "analog health" waiver: their signal rise and fall times are sufficiently good to skip analog circuit analysis and move

[1] See ACL2 home page http://www.cs.utexas.edu/users/moore/acl2/.

from analog to switch level verification. The third sub-part, verifying handshake behavior, is the main focus of this paper.

This last sub-part, *verifying the gate-level transitions against the component's handshake protocol description*, is unique to systems of self-timed circuits[2] because such circuits omit the "clock" that might otherwise provide a global timing reference. Self-timed circuits replace the global clock network that would support synchronous behavior with a distributed network of local handshake protocols to support asynchronous or, more specifically, self-timed behavior. Thereby they also replace setup and hold time constraints between the global clock and local signals with timing constraints between local signals.

The crux of verifying gate-level signals against a handshake protocol is to identify and verify the essential internal timing constraints that make or break the component's protocol description. This task is the subject of this paper.

This paper introduces *ARCtimer*, a framework set up precisely to identify internal timing constraints. ARCtimer targets pattern-based circuit families of handshake components — circuit families that use design patterns to describe the circuit implementations of their components. Families that do so include Micropipeline [5], Tangram and Balsa and Handshake Solutions [6], [7], [8], GasP [9], [10], QDI with precharge buffers [11], [8], [12], Mousetrap [13], and Click [14].

ARCtimer plays a crucial role in the overall design flow of an integrated circuit (chip), but its role comes early in the design process, as part of building a library of handshake components. We run ARCtimer once per library, and use the results over and over again for each and every chip design. Thus, even though ARCtimer plays a crucial role in establishing design correctness, its run times play only a small role in the chip's overall design time-to-market. We therefore have the leisure to "pattern" the timing constraints after the design patterns of the handshake components, making the constraints understandable to the component's designer, easy to maintain, and robust to circuit modifications applied later in the design process.

We have used ARCtimer successfully on the circuit families for Click and GasP, and characterized the "timing patterns" for deterministic, nondeterministic and data-driven handshake components in these families.

The goal of this paper is to build a shared understanding of what a framework like ARCtimer entails, so that others can embellish it or make their own version or improve the underlying tools. The Click and GasP results, relevant though they may be, require a full exploration of the circuits and of the various bundled-data protocols that they use, both of which are outside the scope of this paper. But we will indicate where and how the bundled data and data-driven control fit into the framework, and we will identify related work.

The outline of this paper follows the diagram in Figure 1. Section II explains the context of ARCtimer in the design flow. Section III explains the series of steps a framework like ARCtimer must perform for each handshake component. We distinguish four steps, which are discussed in Sections III-A to III-E. In Section IV, we compare ARCtimer to related work, and summarize what is new. Section V concludes the paper.

II Timing Verification Context

Figure 1 (left-column) shows three stages in a typical chip design flow for self-timed circuits. The stages are marked with the keywords *GUI* (Graphical User Interface), *Parser*, and *STA* (Static Timing Analysis). Other stages, for instance simulation and testing and layout placement and routing, are omitted. Each stage receives information from the yellow-colored center column of Figure 1, called the *Design Library*.

The sub-sections below give a short explanation of these three stages in the chip design flow, the information stored in the Design Library, and their relation to the topic of this paper — timing verification of handshake components.

II-A GUI

Using a GUI (Graphical User Interface) or an equivalent written user interface, one can formulate a network of components connected by handshake channels. The GUI design in the top-left corner of Figure 1 connects four components assembled to generate the Fibonacci sequence 1, 2, 3, 5, 8, etc.

The GUI formulation operates partly at a structural and partly at a functional level, higher than the circuit. Our GUI-formulated designs use function calls to represent data operations. and a handshake protocol based on full and empty channels with data types. A full channel has valid data; an empty channel has data not yet valid or no longer used. The Storage components in the Fibonacci design act when their incoming channels are full and their outgoing channels are empty. When they act, they:

- *copy* the incoming data and forward the copied data,
- *fill* their outgoing channels, making them full, and
- *drain* their incoming channels, making them empty.

The Join component adds the numeric data on its two incoming channels and forwards the sum. Having no storage facility for data, it waits to drain its incoming channels until all its outgoing channels are empty. This ensures that the incoming data remain stable until the sum is stored and acknowledged.

The Fibonacci design starts with all channels empty except for channels *ch1*, *ch2* and *ch3* that start full with initial data values respectively *1*, *1*, and *0* — as indicated in Figure 1. The Join forwards the sum of *0* and *1*, i.e. *1*, both to the *results* channel and to channel *ch4* going into Storage component *C4*. Storage *C4* forwards the Fibonacci result to Storage *C1*, and in doing so it fills *ch5* and drains *ch4*. This enables the Join to drain channels *ch2* and *ch3*, thus enabling Storage *C2* to act. *C2* acts by storing the data value *1* proffered over *ch1* and sending it on to *ch3*, thereby making *ch3* full and *ch1* empty. This in turn enables Storage *C1* to store and forward

[2] In the rest of this paper we will use either of the terms *system*, *design*, and *circuit* to refer to systems designed using circuits.

the new Fibonacci result *1* onto *ch1* and *ch2*, fill *ch1* and *ch2*, and drain *ch5*. The design is now back in a state similar to its initial state, with all channels empty except for *ch1*, *ch2*, and *ch3* that have the next set of data values: *1*, *1*, *1* respectively. The Join's next Fibonacci result will be *2*.

II-B Parser

The Parser takes as input a component network from the GUI and expands it into a gate-level netlist for the protocol and circuit family selected by the user.

For the Fibonacci design in Figure 1, we chose a bundled-data two-phase non-return-to-zero (non-RTZ) handshake protocol, using a request wire, an acknowledge wire, and a bundle of wires with data. The gate-level netlist for Storage *C4*, shown in the center of the left column in Figure 1, belongs to the Click circuit family [14].

There are several choices for expanding data functions, like the *add* function in the Join. One choice is to keep them as function calls. Standard hardware description languages, such as Verilog, can mix structural and functional descriptions [15]. Another choice is to expand the datapath circuits separately and organize the GUI formulation to optimize the flow of data. Standard design compilers excel at automatically synthesizing combinational functions into gate-level netlists. Automatically synthesizing sequential functions is more difficult, but possible when the goal is to optimize worst-case performance. However, a major promise of self-timed design is the ability to optimize average-case performance — in terms of latency, throughput, power, energy, or any combination thereof. Partitioning sequential functions into combinational functions that optimize *average-case rather than worst-case* performance has thus far eluded design automation. Such partitioning remains a collaborative effort between the designer and his or her design compiler [16], [17], [18], [19], [20].

II-C Design Library

An ideal design flow would support a variety of circuit families that could be mixed and matched based on the desired speed, power, energy efficiency, time-to-market or backward compatibility needs for the system or sub-systems. The Design Library for such a flow should store GUI, circuit, and protocol descriptions for the components of each family. Such a library should also store the timing constraints for each component.

The yellow-colored center column of Figure 1 illustrates the Design Library. It shows a Click Storage circuit (top), its protocol (middle), and a static timing analysis (STA) code description of its timing constraints (bottom). Figure 1 omits component, data type, and function descriptions for the GUI. Although the Design Library supports descriptions parametrized for multiple incoming and outgoing channels, the Storage example in the center column has only one incoming and outgoing channel. Section III of this paper uses this single-in single-out Storage component to explain how one can generate timing constraints to fit parametrized components.

II-D STA

Static timing analysis (STA) [21] allows one to validate and repair timing constraints in the gate-level netlist generated by the Parser. Well-known examples of timing constraints for latches and flipflops are minimum clock pulse width, setup time, and hold time. A self-timed Design Library also holds relative timing constraints between end signals on paths that start at the same point but must arrive at their end points in a pre-established sequence. The delay slack in each constraint is parametrized and filled in during technology mapping.

A *Technology Library* for the chosen fabrication process will fill in further details on gate and wire delays, minimum clock pulse widths, etc. By using timing information stored in the Technology Library with physical information obtained from the chip, STA tools can compute and compare actual clock pulse widths against required minimum clock pulse widths, and add extra delay to repair inadequate pulse widths. The repairs go into the next chip layout iteration. STA tools can also repair relative timing constraints by adding sufficient delay to the "late path" with the pre-established later arrival.

There are several STA decisions that one must make, each with its own choices. Below, we will emphasize three important STA decisions, and indicate the choices that we have made.

The first STA decision to make is *where to insert delay* to repair an invalid timing constraint. One could insert the delay at the end point of the pre-established later end signal. Alternatively, one could insert the delay at a design-friendly location that might be exercised less frequently per protocol cycle and therefore retard the circuit performance less. Or one could choose a repair point that is shared by multiple invalid constraints, thus reducing the need to insert multiple delays.

We have chosen to specify a design-friendly delay insertion point for each timing constraint. Each constraint stored in our Design Library identifies a delay insertion point to use for its repair. The Design Library may indicate that the delay is symmetric or that it retards only rising or only falling signals.

We formulate timing constraints from the viewpoint of a handshake component, even though the constrained paths may start or briefly wander outside the component. The STA code for a timing constraint stored in the Design Library records when and where a constrained path enters and exits the component. The "when" relates to a pre-established path signal sequence. The "where" is always a handshake signal because all components connect only through handshake channels. The STA code can identify a constraint with an external start point by identifying the two handshake signals that jointly started there.[3] Armed with this information, an STA tool can instantiate the STA code stored in the Design Library, fill in the sub-paths that are outside the component instance in the

[3] We use this, for instance, to formulate bundled-data setup time constraints. Data flipflop *FF_D* in the Storage component in Figure 1 (center-column-top) has a setup time constraint with an external start point identified as the point where handshake signals *in1_R* and *in1_D* jointly started.

gate-level netlist, and complete the path-finding process in a modular fashion.

The second and equally important STA decision to make is *when to insert delay*. The many timing constraint instances associated with a gate-level netlist might not be independent to each other. Inserting delay to repair one invalid constraint instance may repair or invalidate others.

We use an iterative process similar to [14] for delay insertion. During STA, we group timing constraint instances that share the same delay insertion point instance[4] for repair. For each delay insertion point and its group of constraints, we maintain:

- a list with delays of the constraints in the group,
- the maximum delay in the list, and
- the sum of the delays in the list.

The delay value of a constraint indicates the least delay one must insert into the gate-level netlist to make the constraint valid. The STA process stages delay insertion iteratively, inserting more delay at only one insertion point per iteration. As mentioned earlier, constraints are not necessarily independent, and so inserting more delay into the netlist to repair one constraint may repair others as well, or possibly damage them. Therefore, after each iteration the STA process re-computes the delay requirements for all constraints. The process is as follows:

- Start the first STA iteration, with all delays set to zero.
- After each iteration, update the information for each insertion point. For valid constraints, set the delay to zero. For invalid constraints, set the delay to a re-computed minimum delay mismatch rounded up to the best suitable delay device available in the Technology Library.
- If all groups have a maximum delay of zero then all timing constraints are satisfied, iteration ends, and the netlist can proceed to the next stage in the chip design.
- If one or more groups have non-zero delay, another iteration begins by adding delay to the worst offender. As worst offender, our process chooses an insertion point from those with the highest delay sum. The added delay is the maximum delay listed for this worst offender.

This process is not necessarily monotonic in the number of valid constraints, but it will converge unless constraints are circularly dependent, which rarely happens. Circularly dependent constraints force one to choose different delay insertion points or even different timing constraints.

Having discussed *where to insert delay* and *when to insert it*, we now come to the third and most important STA decision to make: *what STA engine to use*. Conventional STA tools are difficult to use on self-timed circuits because such tools fail to handle logic loops gracefully. Simple treatment of such loops is acceptable for the conventional design process because they are rare in clocked systems — loops in clocked systems tend to start and end at flipflops. Self-timed circuits, however, are rich with logic loops, as they must be because the unstable behavior of closed logic loops animates self-timed behavior.

Graceful analysis of rise and fall times and delay of gates in logic loops requires a two-pass process. A first pass computes output rise and fall times from gate size, gate load, and input rise and fall times. This pass converges very quickly because output rise and fall times are a very weak function, almost independent, of input rise and fall times. A second pass computes the delay of each gate using the input rise and fall times from the first pass.

Conventional STA tools combine those two passes into one concurrent process. They split loops into linear acyclic paths to make a one-pass estimation effective. Moreover, they commonly use a "clock," rare in self-timed circuit designs, to guide where to split each loop. Some self-timed design groups have invested heroic effort in fresh ways to split loops in order to apply conventional STA tools to self-timed systems [22], [23], [14], [24], [25], but none work truly gracefully.[5]

The time has come to use a two-pass process to analyze loops intact. Loops are, after all, central to self-timed circuit design.

Our STA engine is set up to work self-standing or with an existing STA tool. Its internal algorithms to find paths and calculate path delays are still too coarse-grained to replace existing STA tools, but adequate for early design exploration. We use the STA engine in self-standing mode to evaluate the timing in new handshake components before we have formalized timing constraints using ARCtimer — the timing verification framework discussed in Section III. We use the self-standing mode again to validate the STA code for the timing constraints produced by ARCtimer and stored in the Design Library. By inserting pseudo-random delays at multiple pseudo-randomly selected points in the netlist we force the STA engine to recompute compensating delays, and then we simulate and test the repaired netlist for correct functionality. The Click Storage constraints in Figure 10 of Section III-C1 have been validated in this way for 15743 "pseudo-random" test cycles.

II-E Summary

The Design Library stores verified components for use in chip designs. The Design Library appears in the center column of Figure 1 because it connects the chip's design flow on the left and the component's timing verification framework on the right. Once verified, a component may be used in many chip designs. Because closed loops are central to self-timed circuits, the time has come for STA tools to avoid splitting loops and instead to analyze loops intact.

[4] We may use "constraint" and "insertion point" when it is clear from the context that we mean "constraint instance" and "insertion point instance."

[5] This applies also to the Click self-timed circuit family, which was developed specifically to work with conventional STA and test tools [14]. Click circuits use only flipflops as state-holding elements, and have a flipflop in every loop. Some Click loops, however, go *through* flipflops and fail to start or end at flipflops. Conventional STA tools require splitting such loops.

III Timing Verification Framework

The spiral in Figure 1 shows the four main steps in our timing verification framework[6] for handshake components. We call this framework *ARCtimer*. The steps use the keywords: *Handshake Component* (Step 1), *Model Checker* (Step 2), *Timing Patterns* (Step 3) and *Static Timing Analysis* (Step 4). Step 1 begins and Step 4 ends in the yellow center column with the Design Library of component descriptions for each circuit family supported by the design flow. This paper illustrates the steps for a Click Storage component with single incoming and outgoing channels. This same component appears in subsequent sub-sections of this paper to explain each step.

We use this framework in two ways, with and without priming. Without priming, ARCtimer takes the circuit and protocol descriptions of a component and helps us uncover all the timing constraints. The set of timing constraints thus produced ensures that the circuit obeys the protocol. ARCtimer works well without priming for simple components such as the Storage and Join in the Fibonacci design in Figure 1 (left-column). For complex, nondeterministic, or data-driven components the run time and space limitations of underlying tools may necessitate priming ARCtimer with a starter set of timing constraints and using ARCtimer to complete the set.

The sub-sections below explain each step in more detail.

III-A ARCtimer Step 1 — Handshake Component

A handshake component responds to the full and empty state of its channels, as we illustrated earlier in Section II-A for the Storage and Join components in the Fibonacci design.

The circuit-level representations for full and empty channels depend on the variant of the handshake protocol used. Many circuit families, including Click [14], Micropipeline [5], and Mousetrap [13] use a two-phase non-return-to-zero (non-RTZ) protocol with separate request and acknowledge wires to encode full or empty. GasP uses a two-phase return-to-zero (RTZ) protocol [9], [10] with a single statewire to represent full or empty. Figure 2 shows the default representations for full and empty in two-phase non-RTZ and two-phase RTZ handshake protocols.

In general, the control logic of a handshake component is an AND function of the conditions necessary for it to act. Complex handshake components may have multiple such AND functions to guard different actions. The Click Storage component in Figure 1 (center-column-top) has one such AND function — labeled *and2*.

The response of a handshake component usually changes the state of one or more of the channels to which it responded. Many components drain full incoming channels and fill empty outgoing channels. Thus, there is a feedback loop from channel state to component action to channel state. The Click Storage component in Figure 1 (center-column-top) has two such loops: one for channel *in1* from *in1_R* through gates *xor_in1*, *and2*, *FF* to *in1_A*; another for channel *out1* from *out1_A* through gates *xnor_out1*, *and2*, *FF* to *out1_R*.

The AND function coordinates the two loops and makes the Click Storage component "act." The component's gate-level actions are similar but more refined than its GUI-level actions described in Section II-A. The action triggers when *in1* is full ($in1\_R \neq in1\_A$) and *out1* is empty ($out1\_R = out1\_A$) — see Figure 2. When detected, these cause rising transitions on *xor_in1* and *xnor_out1* that in turn cause AND function *and2* to rise. A rising transition on *and2* clocks the edge-triggered flipflops and starts three actions concurrently:

- *FF_D* captures and copies data from *in1_D* to *out1_D*.
- *FF* inverts the value on signal *in1_A*, thus draining *in1*.
- *FF* also inverts the value on *out1_R*, thus filling *out1*.

The now empty *in1* and now full *out1* reset *xor_in1* and *xnor_out1* to low, each of which resets *and2* to low, thus bringing the Click Storage circuit back to an initial state where it can coordinate the next full *in1* and empty *out1* handshakes.

We initialized the Click Storage circuit in the Design Library of Figure 1 (center-column-top) with all channels empty. All its signals have a logical value of 0, except for the output of *xnor_out1* and the *D* input of *FF* which are 1, as indicated.

This initial state in the *Storage circuit* matches the initial state in the compact *Storage Protocol* description in Figure 1 (center-column-middle) and the grey-colored state 0 in the corresponding finite state machine expansion (right column).

One can choose various specification formalisms to describe the protocol behavior of a single handshake component or of a self-timed network of handshake components. Dialects of *Communicating Sequential Processes* (CSP), sometimes called *Communicating Hardware Processes* (CHP), are very popular [6], [8]. The *Calculus of Communicating Systems* (CCS) forms the basis of the self-timed circuit verification work in [23], [26]. *Signal Transition Graphs* and *Petri Nets* form the basis of the self-timed circuit verification and synthesis work in [27], [28], [29].

The goal of this paper is merely to show how to verify single components. We consider here neither how to synthesize a component nor how to verify networks of them. This limited goal gives us the leisure of selecting a formalism whose specifications are both *compact*, i.e. short and easy to understand, and *complete*, i.e. fully delay-insensitive. We found a suitable formalism in the theory of *Delay-Insensitive Algebra* developed by [30], [31], [32]. Delay-Insensitive Algebra also underlies [33] which uses it to build a verification framework for self-timed circuits. Our goal is much simpler than any of the synthesis and verification work built on Delay-Insensitive Algebra. We merely seek compact and complete specifications that allow us to verify that a component's circuit has the same choices of action as specified by the component's protocol. We seek to avoid premature commitment to a verification tool.

[6] We use the term "framework" because we already reserved the term "system" for large-scale designs, and because the term "flow" is often associated with automatic solutions and we seek to avoid that connotation.

Delay-Insensitive Algebra uses finite traces of events that specify not only safety properties, but also liveness properties that are crucial for distinguishing choices of action. It uses an interleaving semantics that represents parallel events by ordering them arbitrarily.

The protocol description in Figure 1 (center-column-middle) first identifies the signals coming into the Click Storage (Input) and those going out (Output). This information will be used to complete the compact description into a fully delay-insensitive one. Next come the handshake event orderings for the two channels. Each event is either a rising or a falling signal transition. Each channel of the Click Storage component starts with an event on its request signal, and thereafter alternates events on its request and acknowledge signals, This corresponds to the basic two-phase non-RTZ handshake communication protocol for an initially empty channel, illustrated in Figure 2. Last comes the protocol description *P* — a compact repetitive sequence of four consecutive input-output events:

$$P = in1\_R\ ;\ in1\_A\ ;\ out1\_R\ ;\ out1\_A\ ;\ P.$$

In this form, protocol *P* says that the Click Storage component must wait for input event *in1_R* before it produces output event *in1_A* followed by output event *out1_R*, after which it waits again until it receives another input event, namely *out1_A*, before it repeats the same protocol, *P*.

The delay-insensitive interpretation of *P* allows more behaviors. The interpretation is based on what is popularly known as the *Foam Rubber Wrapper* metaphor, a term for delay-insensitive communication introduced by the late Charles Molnar. The idea is that an event may be delayed for an arbitrary time when it travels between sender and receiver components. Thus, an input event in an event sequence specified by *P* might have occurred as early as its generation or as late as its receipt, or anywhere in between. Hence, input events *in1_R* and *out1_A* in *P* may move to earlier positions in the sequence provided each input follows the previous output event on the same channel, as specified in the handshake event orderings. Likewise, output events *in1_A* and *out1_R* may move to later positions in the sequence provided each output precedes the next input event on the same channel.

We use tools developed for Delay-Insensitive Algebra in [32] to complete the compact protocol description expressed as *P* automatically into a fully delay-insensitive description expressed as the finite state machine in Figure 1 (right-column). Figure 3 repeats both descriptions.

The finite state machine in Figure 3 (bottom) describes the various event sequences and event choices at the pair of channel interfaces of the Storage component. It also describes the progress expectations at each state in an event sequence. The triangles (▽) denote transient states that may persist only for a finite time. Triangular states typically respond to handshake output events, which are controlled by the component. The underlying assumption is that the internal circuit actions leading up to the output event will finish within a finite amount of time.[7] This is valid for most actions, with the possible exception of non-deterministic arbitration — absent from a Storage component. The rectangles (□) denote non-transient states that may persist forever. Rectangular states typically produce only input events — events controlled by the component's environment. The underlying assumption is that the environment might be lazy and never act. The finite state machine constrains the component to exit a transient state within unbounded but finite time, but allows it to remain in a non-transient state forever.

Note that these descriptions can be used for any Storage component with single incoming and outgoing channels and two-phase non-RTZ handshakes. One can easily envision how to generalize both descriptions to arbitrary numbers of channels. Other handshake components, such as the Join in the Fibonacci design of Figure 1 (left-column-top), and even non-deterministic and data-driven components, also have relatively simple compact descriptions that are easy to understand [32].

The combination of a compact protocol description, *P*, and tool automation to complete *P* into a fully delay-insensitive description helps avoid over-specifying components. Avoiding over-specification is important and harder than one might think. We inadvertently and repeatedly over-specified the handshake behavior of a component using the approach in [23], [26], which requires complete specifications in CCS without tool support to help make them.

Note:
It may be worthwhile to revisit and simplify the theory of Delay-Insensitive Algebra, "building it down" to be just barely expressive enough to describe compact protocols for single handshake components while preserving the ability to complete these descriptions automatically into fully delay-insensitive finite state machines as seen in Figure 3. The simplified theory would be easier to support with tooling and to re-use in other self-timed design and verification flows.

III-B ARCtimer Step 2 — Model Checker

Figure 3 illustrates how one can model the protocol of a handshake component as a finite state machine. The machine serializes sequential as well as parallel events and captures the serialized behavior in event-based state transitions, state transition choices, and transient and non-transient states. Similar finite state machine descriptions can model gates, wires, the network of gates and wires that form the circuit of a handshake component, and even the timing constraints of a handshake component. Verifying that the component's circuit meets the component's protocol under the component's given set of timing constraints thus becomes a model checking task [34].

What model checker should one use for this task? The two basic choices are a general-purpose model checker that is widely used or a model checker customized to fit the self-timed

[7] This assumes that the gates and wires are well-designed, and goes back to designing circuits using the theory of Logical Effort [4] — see Section I.

computation theory of one's choice. *Analyze* and *Artist* in [23], [26] are examples that use customized model checkers with a trace semantics and a CCS based logic conformance relation. They model and verify that the timing-constrained circuit meets the protocol. *Process Spaces* and *FIREMAPS* in [33], [35] are examples that use the theory of Delay-Insensitive Algebra for both the modeling and the model verification task.

A major advantage of a customized model checker is that the theory is already built into the model checker. For instance, a model checker built on Delay-Insensitive Algebra can use the finite state machine description in Figure 3 (bottom) directly. On the other hand, a customized model checker tends to have few and highly specialized users and few test examples, and may be flawed by various subtle bugs that make it hard to use for new examples. A major advantage of a widely used general-purpose model checker is that it has many users and many diverse test examples, and so its bugs tend to be discovered and repaired.

We experimented with both customized and general-purpose model checkers. We have found customized model checkers especially hard to use for modeling and verifying the protocols and circuits of non-deterministic and data-driven handshake components. Moreover, we mistrusted some of the verification results that we obtained. We resolved the difficulty in modeling the protocols by using formalisms and tools developed for Delay-Insensitive Algebra, as explained in Section III-A. Other difficulties vanished with use of a general-purpose model checker. General-purpose model checkers force one to indicate explicitly both what to verify and how to execute the various parts of a model. Although explicitness requires more work, it gives one full control over one's own experiments.

The experiments and code fragments reported in this paper are based on NuSMV [36], a model checker that is freely available and has an active and diverse user community. NuSMV has helped us generate and verify timing constraints for widely different components with deterministic, non-deterministic, and data-driven handshake behaviors. The timing verification work in [37] each use NuSMV but verify fewer properties than we do, as we will explain in Sections III-B1–III-B2 and IV.

Figure 4 shows what a general-purpose model checker must have and do to verify a handshake component's circuit against its protocol under a given set of timing constraints. Note that besides models for the circuit, protocol, and timing constraints, there is the component's environment — a model for the environment in which the component's circuit operates. We model the component's environment by providing a separate interface for each channel that responds to channel outputs in any of all the valid ways possible for that channel.

The following sub-sections give a more detailed explanation of Figure 4, including code fragments with NuSMV solutions.

III-B1 Modeling the Component's Protocol:

Figure 5 repeats the complete, fully delay-insensitive protocol specification of Figure 3 (bottom) and shows its translation into NuSMV model checker lingo.

The translation is wrapped in a self-contained module, with the abbreviated name *protocol*, with formal parameter names for the handshake signals. The module's full name is *Click_Storage_1_In_1_Out_Protocol*.[8] We store such modules in the Model Checker Library — see Figure 4.

The first part of the translation, up to line 26 in Figure 5, codes the states, initial state, and event-based state transitions of the protocol. Each translated state name begins with the letter *s* followed by the original state number — e.g. initial state *0* (top) translates to *s0* (bottom). The original protocol specifications in Figure 3 specify only legal states and transitions, omitting illegal and irrelevant ones. The omissions must be coded, however. We code two types of error states to receive illegal handshake transitions: illegal channel outputs go to *errorOUT*, and illegal channel inputs go to *errorIN*. All other events, irrelevant to the protocol, preserve the protocol's state. The resulting code forms a monitor. It will be used to monitor the *Component's Test Circuit* — the sub-system inside the broken line in Figure 4 (middle) which holds the component's circuit, environment, and timing constraints.

To monitor the Component's Test Circuit the protocol operates in synchronous mode, as we already mentioned in Figure 4. This means that the protocol's finite state machine code is executed in each execution step by the model checker. NuSMV uses the keyword *TRANS* in line 6 of Figure 5 to indicate that the next statement is to be executed in synchronous mode. The next statement, enclosed by the keywords *case* and *esac* in lines 7 and 26, is precisely the monitor code of the component's protocol in the rightmost white rectangle in Figure 4.

The purpose of monitoring the Component's Test Circuit is to annotate its behavior for verification. Verification is done by checking properties. The properties in lines 28–50 of Figure 5 specify what the protocol must see when it monitors the Component's Test Circuit.

The properties in the second part of the code, lines 28–50, are inherent in the protocol specification, and translated along with the rest of the code. The two *safety* properties in lines 30–31 allow only legal handshake behaviors. The five *progress* properties in lines 33–37 allow the five transient states to persist for only a finite time. The transient states correspond to the triangles ($\bigtriangledown$) in the original specification. The remaining *choice equivalence* properties spell out the choices of action that must be available to the observed sub-system to meet the protocol specification. These might be refined with additional event information, if needed. The structure of these properties is quite straightforward for the Click Storage component, but becomes more interesting for non-deterministic components.

The progress and choice equivalence properties are absent from the NuSMV based timing verification work in [37]. We

[8] Its un-abbreviated name says that the module has the protocol translation for a Click Storage component with 1 incoming and 1 outgoing channel.

will come back when we compare related work in Section IV.

III-B2 Modeling the Component's Circuit and Environment:

Figure 6 repeats the gate-level Click Storage circuit and environment models in Figure 1 (right-column-top) and shows the corresponding gate-level NuSMV translation, using two gate models, *cgate* and *ff_posedge* that are defined in Figure 7.

The two translations are wrapped in self-contained modules with formal parameter names to support the exchange of handshake and timing constraint signals. For this paper, we abbreviated the module names *Click_Storage_1_In_1_Out_Circuit* and *Click_Storage_1_In_1_Out_Environment* to *circuit* and *environment*, respectively. These contain the code for the middle two white rectangles with horizontal text in Figure 4.

The *Click Storage Circuit* shown in Figure 6 (top) contains five more buffers than the original circuit description in the Design Library of Figure 1 (center-column-top). The extra buffers are colored grey and named *buf_in1_A1*, *buf_in1_A2*, *buf_out1_R1*, *buf_out1_R2*, and *buf_ck*. The translation adds these buffers to delay wires and individual wire branches independently from gates. Buffers are necessary because the model checker ignores wire delays. Adding a buffer or inverter device to a logical wire connection makes that connection visible to the model checker as a device output with a device delay. It suffices to add buffers only to wires that branch out and to wires that clock edge-triggered flipflops. It is straightforward to adapt a compiler that generates the original circuit description to generate also the description for the model checker. The circuit description for the model checker also contains datapath signals *in1_D* and *out1_D* and datapath flipflop *FF_D*, which are omitted from Figure 6 because they are outside the scope of this paper.

Figure 6 (bottom) shows the translated circuit module in lines 1–18 and the translated environment module in lines 21–30. Lines 3–12 describe the gate instances and their connections for the circuit. Lines 23–24 do the same for the environment. Most gates are instantiated as *process cgate (function,...)* where *function* is a Boolean logic combination of the module's parameters and outputs of other gates. The instances have the same names and logical functions as in Figure 6 (top). For example, gate instance *xor_in1* in line 3 computes the exclusive-or of parameter *in1_R* and *buf_in_A2.val*, the output of gate *buf_in_A2*. Likewise, positive edge-triggered flipflop instance *FF* in line 7 copies and stores the value on *inv_q2d.val* onto its output *q* whenever its clock input *buf_ck.val* changes from low (FALSE) to high (TRUE). The signal definitions in lines 14–17 and 26–29 following the keywords *DEFINE* serve to shorten and simplify various code fragments.

The operations of the circuit and its environment are monitored by the protocol, as explained in Section III-B1. Because we describe protocols with Delay-Insensitive Algebra, which uses an interleaving semantics, the protocol model interleaves its events. Thus, the protocol can interpret handshake events only when they arrive in sequence. Consequently, the circuit and its environment must interleave all handshake events because these are the events they share with the protocol. To simplify the overall execution, we chose to interleave not just the handshake events but all events generated by *cgate* instances in the circuit or its environment.[9] NuSMV pairs the keywords *process* and *cgate* in lines 3–12 and 23–24 to indicate that the *cgate* instance is to be executed in asynchronous mode by interleaving its operations with those of other *process cgate* instances.

The asynchronous interleaving mode of operation comes with a cost of fairness conditions for selecting which *process cgate* operation to run next. The protocol assumes that most circuit operations take a finite time. It expects the circuit to generate a handshake output within a finite number of execution steps after receiving a handshake input from its environment. The NuSMV code for the protocol uses progress properties to formulate and verify these expectations — see lines 33–37 of Figure 5. To satisfy these progress properties, each *process cgate* instance in the module must be selected to run after every so many unbounded but finite execution steps. The NuSMV statements *FAIRNESS running* in lines 18 and 30 of Figure 6 enforce precisely that.

The remaining code details can be explained by examining the module definitions for *cgate* and *ff_posedge* in Figure 7.

The module definition of *cgate*, i.e. "combinational gate," follows in lines 1–20 of Figure 7. Each *cgate* takes an arbitrary Boolean combinational logic function through its first parameter, *set*. For example, the *cgate* for *xor_in1* in line 3 of Figure 6 takes *in1_R xor buf_in1_A2.val* — the exclusive-or of Boolean signals *in1_R* and *buf_in_A2.val*. The second parameter, *init_val*, contains the initial value of *cgate* output *val*, assigned in line 6 of Figure 7. For example, the output of *xnor_out1* in line 4 of Figure 6 is initialized to TRUE, which corresponds to the value *1* indicated for the *xnor_out1* output in Figure 6 (top). When a *cgate* instance is selected to run, it evaluates its *set* function. Depending on the other input parameters in Figure 7, it either updates its output *val* with the *set* result (line 10 or 11) or does nothing (line 9 or 10). Only lazy or timing constrained *cgate* instances may do nothing.

A *cgate* is lazy if its third parameter, *lazy*, is TRUE. For example, both the Click Storage Environment gates *ENV_in1* and *ENV_out1* in lines 23–24 of Figure 6 (bottom) are lazy. A lazy *cgate* has an arbitrary choice either to act by setting its output *val* to the result in *set* or to do nothing by keeping the old value of *val*. This nondeterministic choice is indicated in line 10 of Figure 7 by the curly brackets around *val* and *set*.

Timing constraints may prevent a *cgate* output transition from FALSE to TRUE (rise), from TRUE to FALSE (fall), or both.

[9] This simple mode of interleaving can be combined with a *simultaneous* mode of operation [37] for internal gates that generate non-handshake events, allowing arbitrary subsets of these to operate simultaneously. Such a simultaneous mode of operation would, however, require tighter fairness conditions than the *FAIRNESS running* in lines 18 and 30 of Figure 6 (bottom) in order to satisfy the protocol's progress properties in lines 33–37 of Figure 5.

Output *val* cannot rise in line 9 of Figure 7 if the fourth parameter *stop_rise* is TRUE, and neither can it fall if the fifth parameter *stop_fall* is TRUE. In Section III-C, we will discuss how timing constraints control the run-time values of *stop_rise* and *stop_fall* in the various *cgate* instances.

It is possible that a *cgate* instance, poised to have its output rise or fall, fails to be selected and do the output transition before a new *set* value arrives that disables the transition. For *cgate* instances used in self-timed circuits, the presence of a later *set* value overtaking an earlier one often indicates the presence of a race condition. We therefore flag such overtakings for later inspection. A variable with the name *semimodular*, initially TRUE (line 7), becomes FALSE at the first such overtaking (lines 15–16) when the next execution step no longer shows an enabled transition (*next(val)=next(set)*) but also shows no sign of having taken it (*next(val)=val*). The NuSMV model checker updates variable *semimodular* (lines 14–18) at each execution step, as indicated by the keyword *TRANS* in line 13. The "digital health" property in line 20 requires *semimodular* to be TRUE at all times, and flags any change to FALSE.

Variable *semimodular* in Figure 7 has been aptly named. *Semimodularity* is a well-known paradigm for designing self-timed digital circuits without hazards by insisting that digital signal changes occur before being disabled. One might call it the "no change left behind" paradigm. Introduced by David Muller [38] and brought to the attention of a wider audience through Raymond Miller's 1965 book [39] semimodularity formed the starting point of the first generation of self-timed circuit design tools [27], [28]. Though semimodularity is still an important paradigm for designing and verifying self-timed circuits, new design trends for fast, energy-efficient self-timed circuits [18], [40], [19], [20] force it to share that position with *Relative Timing* [41]. The NuSMV code in lines 14–18 of Figure 7 for updating the variable *semimodular* is based on a new definition of semimodularity for timing constrained self-timed circuits, presented by us in [42].

The module definition of *ff_posedge* in lines 22–31 of Figure 7 models a positive edge-triggered flipflop. The flipflop copies and stores the value of its second parameter, *d*, onto its output, *q*, whenever its first parameter, *ck*, changes from low (FALSE) to high (TRUE), as indicated in line 29. The value of output *q* is initialized through the third parameter, *init_q* (line 26). Instances of *ff_posedge* run each execution step, as indicated by the keyword *TRANS* in line 27.

To time and verify each *ff_posedge* instance, we pair it with a *process cgate* instance as its clock buffer. The clock buffer provides the timing flexibility in selecting when the flipflop acts. We verify the semimodular behavior of the clock buffer to ensure that all "clock" transitions issued by the *and2* gate reach the flipflop — see Section III-A for a reminder on "clocking." This explains the extra buffer *buf_ck* in Figure 6 (top): it is the clock buffer for flipflop *FF*. It is possible to model *ff_posedge* so it can be timed and verified without the extra clock buffer, but to do this properly requires modeling data — which falls outside the scope of this paper.

Gate models *cgate* and *ff_posedge* in Figures 6–7 have NuSMV code descriptions reminiscent of code descriptions in a hardware description language like Verilog. We chose to use a general gate model for *cgate*, capable of modeling all combinational gates in the Click Storage component. This is possible because each gate instantiated in the component's gate-level netlist in Figure 6 (top) has a behavioral description of its Boolean logic function. When instantiated with the signals coming into the gate, this Boolean logic function becomes the *set* function of the corresponding *cgate* instance in Figure 6 (bottom). One could follow a similar approach for sequential gates and define a general gate model capable of modeling all sequential gates, as is done in [43]. We refrained from doing this here because Click components use only one type of sequential gate — a positive edge-triggered flipflop. Instead of using a few general gate models, one could define a dedicated model for each gate with a different logic function, and connect the gates by connecting their signal names. This is done in [37]. Figure 6 would require eight such dedicated gate models: two for the lazy environment, and six for the circuit. Dedicated gate models produce a larger Model Checker Library to characterize, but they contain extra connectivity information that could be useful.

III-B3 Instantiating the Models in a Model Checker Netlist:

Figure 8 repeats the middle grey rectangle of Figure 4 with the Model Checker Netlist but omits the white rectangle for the Component's Timing Constraints. It also shows the NuSMV translation with a single protocol, circuit, and environment instance for each. The keywords *process* in lines 4–5 indicate that the model checker will run the circuit and environment instances in asynchronous mode by interleaving their events. The *FAIRNESS running* command in line 11 insists that the event selection between the two instances be fair. The lack of keyword *process* in line 3 indicates that the protocol instance runs in synchronous mode. This matches the modes of operation specified earlier in Figure 4.

In Section III-C, we will verify the "digital health" and protocol properties in the code of Figure 8, analyze any failing properties, and generate timing constraints to correct the failures. In Section III-D we will revisit Figure 8 and upgrade its NuSMV code by adding the missing constraints.

III-C ARCtimer Step 3 — Timing Patterns

When the model checker runs the code in Figure 8 it reports multiple failing properties. For each failing property it gives a counterexample — a computation path that fails that property. Failing properties expose delay sensitivities in the design. A counterexample not only exposes a delay sensitivity, but also contains "clues" about how to prevent it from becoming hazardous. These clues can be captured in a form suitable for verification and correction — and thus prevention.

There are various options available for capturing clues. For instance, [44] assigns metric delay bounds to each gate in the circuit and its environment, capturing each clue as a tighter metric delay bound and calls this a *timing constraint*. Alternatively, a clue can be captured as a relative ordering of events and be called a *chain constraint* as in [33], [35], or a *(relative) timing constraint* as in [45], [29], [46], [23], [26].

Here, we capture a clue as a relative ordering of events and call this a *relative timing constraint*, or simply *constraint*.

Analyzing a counterexample to capture the clue it contains always requires finite state machine analysis around the failing step. Many of the approaches referenced here, notably [44], [46], [29], [47], [26], provide heuristics to capture the clue as a constraint and to generate the constraint automatically. These heuristics are, alas, tightly coupled to the underlying tools and theory and thus hard to transfer to other verification flows.[10]

To share understanding of what is involved in analyzing a counterexample, this paper analyzes the two counterexamples of Figure 9 for the netlist of Figure 8 — one failing a "digital health" property and the other failing a protocol property. Section III-C1 analyzes each counterexample, extracts its clues and formulates them as relative timing constraints. Section III-C2 shows a way to model these constraints.

It is well to remember that the generation of relative timing constraints comes early in the design process, as part of building the library of handshake components — the Design Library in Figure 1 (center-column). Once constraints are known and stored in the Design Library, they are used over and over again for every chip design. Thus, the time taken for constraint generation plays only a small role in the overall time from design to market. We therefore have the leisure to make the constraints understandable to the component's designer, and to increase their robustness to circuit modifications applied later in the design process. We do this by formulating the constraints as *timing patterns*, in support of the *design patterns* that the designer selected for the component's circuit and family. The highly general and highly robust timing patterns derived for simple components can form a starter set for *priming* complex components. More detail on timing patterns appears in Section III-C3.

III-C1 Analyzing Counterexamples:

Figure 9 (right) shows two counterexamples for the NuSMV netlist in Figure 8. To ease following the paths in each counterexample, Figure 9 also repeats the gate-level circuit diagram of the Click Storage circuit and environment. Both counterexamples describe a path of events starting from the initial state. State names, like *s0* for the initial state, are filled in by the component's protocol — the vertical rectangle in the netlist diagram of Figure 8. The protocol description for the Click Storage component can be found in Figure 5.

The two counterexamples show that if gates and wires have arbitrary delays it is harder to guarantee the simple operational descriptions of handshake components and handshake channel interfaces given in Section III-A and Figure 2.

The first six execution steps, run steps 1–6 in Figure 9, are the same in both counterexamples. In step 1, *ENV_in1* raises *ENV_in1.val*. The rising transition is denoted by the symbol "$+$" at the end of *ENV_in1.val*. For a falling transition we would have used the symbol "$-$." Remember that a gate name with suffix ".val" denotes the gate's output — see Figures 6–7. Because *ENV_in1.val* is an alternative name for *in1_R*, this step changes the protocol state to *s1*. With *in1_R* high and *in1_A* still low, incoming channel *in1* is now full. This is detected by gate *xor_in* whose output rises in step 2. With both its input signals high, AND function *and2* now "acts" as follows. First *and2.val* rises (step 3), and then clock buffer output *buf_ck.val* rises and clocks flipflop *FF*, causing its output *FF.q* to rise (step 4). From here on, the ordering of execution steps depends on the delays of the logic gate in the feedback loops from *FF.q* back to *buf_ck*. There are four such feedback loops — two per channel, on each side.

Both counterexamples focus on the two feedback loops at the *out1* side. They show what happens when the two feedback loops are equally fast, and what happens when both are faster than the two feedback loops at the *in1* side. The examples both next select to change *buf_out1_R1* in step 5, raising *buf_out1_R1.val* and thus *out1_R*, which changes the protocol state to *s3*. Outgoing channel *out1* is now empty. In step 6, both examples then raise *buf_out1_R2.val*, making gate *xnor_out1* aware that *out1* is empty by enabling *xnor_out1.val* to fall. In step 7, the two counterexamples diverge.

The example in the left box of Figure 9 selects *ENV_out1*, raising *ENV_out1.val* and thus *out1_A*, which changes the protocol state to *s4*. The change in *out1_A* also makes channel *out1* full and prevents *xnor_out1.val* from falling before it took the opportunity to fall. This causes *xnor_out1.semimodular* to become FALSE (lines 15–16 of Figure 7) which is flagged because the gate has failed the "digital health" property, called *CTLSPEC AG semimodular* (line 20 of Figure 7).

A semimodularity failure like this could happen in a chip design if the internal path through the circuit, from *FF.q* via *buf_out1_R2* to *xnor_out1*, were to take about the same time as the external path through the environment, from *FF.q* via *buf_out1_R1* to *xnor_out1*. Were this to happen, it would render exclusive-NOR gate *xnor_out1* useless as a detector of full and empty channel states, thus defeating the handshake protocol on *out1*. To differentiate a full from an empty channel, *xnor_out1* must have enough time to receive and respond to the internal representation for *out1_R*, as captured by *buf_out1_R2*, before the environment responds with a next state change through *out1_A*. This is the clue we are seeking. Given that both inputs for *buf_out1_R1* and *buf_out1_R2* start

[10] (See also *Note* on page 6 in Section III-A)
It may be worthwhile to revisit the existing heuristics on *automatic generation of constraints from counterexamples* and "build down" the surrounding theory to be just expressive enough for heuristic constraint generation. The extracted heuristics would be easier to support with tooling and to re-use elsewhere.

at FF.q, or even at the AND function *and2.val* before that, we can capture this clue in the counterexample in one of the following two ways:

- After *FF.q* rises, *xnor_out1.val* must fall before *out1_A* rises. Following the notation of [23], [37] we denote this as: $\mathit{FF.q}+ \rightarrow \mathit{xnor\_out1.val}- < \mathit{out1\_A}+$.
- If $\neg \mathit{FF.q}$ holds while *and2.val* rises, then subsequently *xnor_out1.val* must fall before *out1_A* rises — denoted as: $(\neg \mathit{FF.q} \wedge \mathit{and2.val}+) \rightarrow \mathit{xnor\_out1.val}- < \mathit{out1\_A}+$.

The second formulation of the captured clue matches relative timing constraint *rt3* in Figure 10. A similar counterexample exists for the case that *FF.q* holds while *and2.val* rises, leading to *rt4*. Two more such counterexamples can be found by exchanging the two feedback loops at channel *out1*'s side for the two feedback loops at channel *in1*'s side. The four relative timing constraints *rt1* to *rt4* in Figure 10 block all such counterexamples.

Semimodularity failures are easy to solve: instead of disabling the transition, take it! This simple heuristic, however, tends to push the semimodularity failure to the next gate, just as a snow plow pushes snow elsewhere. This happens for instance between *rt7*–*rt8* and *rt9* in Figure 10, each of which solve a semimodularity failure. Constraints *rt7* and *rt8* solve a semimodularity failure for gate *and2* by pushing the failure to the next gate, *buf_ck*. Constraint *rt9* solves the semimodularity failure for gate *buf_ck* by pushing the failure to *FF*, which does not register this type of failure, and so the simple heuristic snow plow stops here. Relative timing constraints that merely push a semimodularity failure elsewhere fail to be appealing and intuitive to the designer of the component and are less robust to circuit modification applied later in the design process. We will come back to this in Section III-C3.

The counterexample in the right box of Figure 9 avoids the mistake of the first counterexample by taking the still-enabled transition $\mathit{xnor\_out1.val}-$ (step 7). It continues by resetting the AND function and setting up the flipflop for the next handshake coordination (steps 8–10). So far so good. But then, it starts a second handshake on channel *out1* (steps 11-12) while ignoring the still outstanding first handshake on *in1* — forgetting that it "takes two to tango." With *in1_R* high and *in1_A* still low, input channel *in1* is still full and *xor_in1* is still high. As a result, the AND function "acts" prematurely (steps 13–15) and coordinates the first handshake on *in1* with the second handshake on *out1*. This premature action of the AND function causes a protocol failure in step 15.

This second counterexample violates the core purpose of the Click Storage component, which is to coordinate exactly one incoming handshake with exactly one outgoing handshake and to repeat this for the successive handshakes on each channel. For one-to-one coordination, the AND function must know when a channel is willing to participate ("Shall we dance?") as well as when its participation is over ("Thank you!"). Each channel indicates its willingness to participate by raising the output of its exclusive-(N)OR gate, and each channel ends its participation by lowering this same output. After each action, both outputs must fall before either rises again. We capture this clue in the counterexample in the following way:

- When *and2.val* rises, then *xor_in1.val* must fall before *xnor_out1.val* rises. We denote this as:
  $\mathit{and2.val}+ \rightarrow \mathit{xor\_in1.val}- < \mathit{xnor\_out1.val}+$.

This formulation of the captured clue matches *rt5* in Figure 10. The related constraint, *rt6*, avoids similar counterexamples for the reverse situation by preventing each handshake on *in1* from outpacing its handshake partner on *out1*.

Solving protocol failures may be hard and require rules of thumb for designing self-timed circuits. For example [29] experiments with slow versus fast input events to guide the synthesis of self-timed circuits. By presuming a slow environment, it may be possible to generate *rt5* automatically from the second counterexample. We will come back to this in Section III-C3.

III-C2 Modeling Relative Timing Constraints:

The relative timing constraints in Figure 10 capture the clues from the various counterexamples generated by the model checker. The two counterexamples of Figure 9 gave us *rt3* and *rt5*, and implicitly all six constraints, *rt1* to *rt6*. The focus of the current section is to expose the structure and operation of all such relative timing constraints.

As mentioned in Section II-D, relative timing constraints are constraints between signals at the ends of paths that start at the same point — signals that must change in a pre-established sequence. Each relative timing constraint identifies the point where the paths split, called a *Point of Divergence* (POD) in [23], [37] — here we call it *myPOD*. Each constraint also indicates the two destinations, a pre-established "early" end point and a pre-established "late" end point — we call these *myEARLY* and *myLATE*, respectively. In addition the constraint has a name, like *rt1* in Figure 10 — we call this *myNAME*.

Our relative timing constraints have the following structure:

- $\mathit{myNAME} : \mathit{myPOD} \rightarrow \mathit{myEARLY} < \mathit{myLATE}$

where

- *myPOD* is an abbreviation for: $\underline{\mathit{guard}_{POD}} \wedge \mathit{event}_{POD}$
  - $\underline{\mathit{guard}_{POD}}$ is a guard, i.e. a Boolean logic expression
  - event_{POD} is an event, i.e. a rising or falling signal
- *myPOD* holds if and only if
  - event_{POD} occurs, and
  - meanwhile $\underline{\mathit{guard}_{POD}}$ holds
- *myEARLY* and *myLATE* have similar structures:
  - *myEARLY* abbreviates $\underline{\mathit{guard}_{EARLY}} \wedge \mathit{event}_{EARLY}$
  - *myLATE* abbreviates $\underline{\mathit{guard}_{LATE}} \wedge \mathit{event}_{LATE}$

The better to distinguish guards from events, we underline guards. We omit trivial guards, like TRUE. For instance, the guards for *myPOD* in *rt5* to *rt11* are omitted for this reason.

Constraint $\mathit{myNAME} : \mathit{myPOD} \rightarrow \mathit{myEARLY} < \mathit{myLATE}$ says:

- if *myPOD* becomes valid
- then *myEARLY* must become valid
- before *myLATE* becomes valid.

One can use a constraint for analysis and report whether or not it is satisfied for all possible computation paths of the system. This is done, for instance, during static timing analysis — see Section II-D. Alternatively, one can use a constraint as an actuator — a delay device that retards $event_{LATE}$ after *myPOD* becomes valid by blocking $event_{LATE}$ until *myEARLY* has become valid. The model checker uses constraints as actuators.

Our model checker's actuator model of constraint *myNAME* is a three-state finite state machine extension of the two-state version used in [37]. The three states are necessary for modeling the non-trivial guards of *myLATE* in *rt7* and *rt8* of Figure 10. We name the three states GREEN, YELLOW, and RED.

Figure 10 (left) shows the *stoplight* model that we use as the model checker's actuator view of a relative timing constraint. Both GREEN and YELLOW states permit $event_{LATE}$ to happen, while a RED state blocks $event_{LATE}$. Most constraints start in GREEN, as do *rt1* to *rt11* in Figure 10 (right), and proceed as follows:

- All constraints go to a GREEN state when *myEARLY* becomes valid, because the need to retard $event_{LATE}$ vanishes with arrival of *myEARLY*.
- In GREEN, only *myPOD* can change the state, because *myEARLY* and *myLATE* matter only after *myPOD* becomes valid. The stoplight changes from GREEN to YELLOW if *myPOD* holds but $guard_{LATE}$ does not. Only instances of $event_{LATE}$ for which $guard_{LATE}$ holds need blocking. The state changes from GREEN to RED if both *myPOD* and $guard_{LATE}$ hold.
- Both YELLOW and RED states follow from arrival of a valid *myPOD* but not yet a valid *myEARLY*.
- Before *myEARLY* becomes valid, changes in $guard_{LATE}$ change the state from YELLOW to RED, and vice versa. Such changing of the guard and the state happens in some computations for *rt7* and *rt8* in Figure 10. The stoplight state for *rt7* or *rt8* changes from RED, at *myPOD*, to YELLOW, by *rt5* and *rt6*, back to RED if *xor_in1.val*+ or *xnor_out1.val*+ changes before *and2.val*−.

III-C3 Deriving Timing Patterns:

Failure analysis of the two counterexamples in Figure 9 of Section III-C1 gave us the first six relative timing constraints *rt1* to *rt6* of Figure 10. Constraints *rt1* to *rt6* are the weakest relative timing constraints required to prevent the failures exposed by the two counterexamples and similar examples. The remaining constraints are also the weakest relative timing constraints of their kind:

- Constraints *rt7* and *rt8* form the weakest relative timing constraints to maintain the "digital health" of gate *and2* as a semimodular gate. They go as far as to permit a single rising *and2* input after both inputs have gone low, before they require *and2.val* to fall.
- Constraint *rt9* is the weakest relative timing constraint to maintain *buf_ck*'s "digital health" as a semimodular gate.
- Constraints *rt10* and *rt11* are the weakest setup time constraints for positive edge-triggered flipflop *FF*.

One can explore *myNAME* : *myPOD* → *myEARLY* < *myLATE* expressions to get an idea which constraints are critical. One way to do this is to estimate the elapsed time between *myLATE* and *myEARLY* at full speed operation under reasonable gate delays and in a reasonable environment, e.g.:

- Assume gate delays equivalent to 2 inverter delays for X(N)OR, AND, FF. Assume zero delay for the grey buffers. Replace the component's environment in Figure 6 by two other Click Storage circuits, one on each channel. Assume maximally parallel operation — no stalling.
- Under these assumptions, the cycle time for *and2.val*+ is 12 inverter delays, and the elapsed time from *myEARLY* to *myLATE* is 4 inverter delays for *rt1*–*rt4* and *rt7*–*rt8*, 6 for *rt5*–*rt6* and *rt9*, and 9 for *rt10*–*rt11*.

With at least 4 inverter delays to spare in each constraint, these estimations indicate that the risk for violating a constraint is low and that none of the constraints *rt1* to *rt11* is critical.

Constraints *rt1* to *rt11* are the weakest possible constraints in part because they are tightly coupled to the circuit. A tight coupling between constraints and circuit is useful if the chip uses exactly this circuit for each instance of the Click Storage component — which is unlikely. For example, a technology mapping tool might partition the AND gate into a NAND and inverter, and a layout tool might add clock buffers. With a NAND gate or extra clock buffers, constraints *rt1* to *rt11*, as formulated in Figure 10, no longer suffice because the gate names and connections have changed. To make the constraints suffice might require a grouping of gates in the new circuit and a mapping of group names back to the old circuit. This is common practice and not a problem in itself. The problem is that not all renamings ensure that *rt1* to *rt11* still cover all the properties in the new circuit — see Figure 11.

Ensuring that the renaming works for *rt1* to *rt11* may require re-running the model checker on the new circuit. However, re-running the model checker would defeat the purpose of working with a Design Library of verified components and would put the timing verification framework, i.e. ARCtimer, in the critical design cycle of each chip. Our purpose holds to keep ARCtimer firmly in the early part of the design process.

To hold this purpose, the constraints must work regardless of circuit changes made during technology mapping or layout. Figure 11 shows that constraints *rt1* to *rt11* fail this purpose.

In summary: *We need general constraints that emphasize the circuit's intent rather than the circuit's structure.*

The component's designer faces a similar issue when choosing appropriate structures for the component's circuit. To make the circuit work for every chip, he or she uses design patterns. The patterns work for most technology mappings and layout tools. We wish to solve circuit design and circuit timing in a similar way. We seek *timing patterns* that make the *design patterns* work — i.e. that ensure:

- the X(N)OR gates detect full and empty channel states,
- the AND function coordinates the handshakes, and
- the *FF* and *inv_q2d* flip the channel state.

Let us examine the initial constraints *rt1* to *rt11* of Figure 10 to see which might work as patterns and which need generalizing:

- Constraints *rt1* to *rt4* make the X(N)OR gates work, and do no more and no less than that — they make fine patterns. Figure 12 rephrases them as *p1* and *p2*.
- Constraints *rt5* and *rt6* make the AND function work by comparing the output signals of the X(N)OR gates. This comparison makes less sense for complex AND functions in components with more than one channel on each side. Requiring the outputs of all X(N)OR gates to fall before any channel input changes results in the more general constraints *p3* and *p4* in Figure 12 (top) and *p* (bottom).
- Constraints *rt7* to *rt9* keep the AND function semimodular, but they do this by exposing the organization of the AND function all the way from gate *and2* to the *FF*'s clock input — a result of resolving semimodularity failures by pushing them out of the way. The slow environment presumed in Section III-C1 for *rt5* and *rt6* can be assumed again here to guarantee that there will be enough time to stabilize internal feedback loops up to *FF*'s clock input before the channel inputs change. This assumption is formalized in *p5* and *p6* of Figure 12. Unlike *rt7* and *rt8*, patterns *p5* and *p6* are robust to both post-layout design changes shown in Figure 11.
- Constraints *rt10* and *rt11* keep the *FF* with *inv_q2d* combination flipping, but can be generalized as patterns *p7* and *p8* of Figure 12 by assuming a slow environment.

Note that each pattern in *p1* to *p8* of Figure 12 still leaves at least 2 inverter delays to spare under the earlier estimations for full speed operation, reasonable gate delays, and a reasonable environment. This indicates that the risk for violating one of these patterns is low and that none of them is critical.

The slow environment assumed above leads to a *burst-mode* operation [48] of the Click Storage component, where internal loops stabilize before new external channel inputs arrive. The burst-mode assumption is expressed most clearly in the parametrized pattern *p* of Figure 12. It is quite common in self-timed circuit design to assume that an external feedback loop through the component's environment is slow compared to an internal feedback loop in the component's circuit. Heuristics for automatic circuit synthesis or timing constraint generation often use such assumptions. There is no guarantee, however, that relative timing constraints generated on the basis of heuristics are sufficiently general to be stored in a Design Library for use in every chip design.

The role of ARCtimer's Step 3 is to take the initial timing constraints, obtained by human or automated failure analysis, and turn them into sufficiently general timing patterns.

III-D Step 2 Revisited — Adding Timing Constraints

The double-headed arrow in Figure 1 (right-column), on the spiral between Step 2 and Step 3, indicates that we alternate these two steps. We first run the model checker (Step 2), then we examine a few counterexamples and capture their clues in one ore more relative timing constraints (Step 3). Then we model the constraints, and re-run the model checker primed with these constraints. We examine a few counterexamples, and repeat. We alternate Step 2 and Step 3 until the model checker reports no further counterexamples. This alternation gave us constraints *rt1* to *rt11* of Figure 10, from which we then derived timing patterns *p1* to *p8* and *p* of Figure 12.

The purpose of this section is to illustrate how one can model such constraints in a general-purpose model checker. As before, we use the NuSMV model checker as example. Figure 13 expands the NuSMV Model Checker Netlist of Figure 8 without timing constraints to match the version shown in Figure 4 (middle) with *p1* to *p8* as the component's timing constraints. When the model checker runs the code in Figure 13 it reports no further counterexamples — all modeled properties are valid.

Module *rt* in lines 1–18 of Figure 13 models the state changes of the *stoplight* model of Figure 10 (left) which is our model of a relative timing constraint. The parameters in line 1 with names *eventPOD*, *eventEARLY*, *guardPOD*, *guardEARLY*, and *guardLATE* represent respectively $event_{POD}$, $event_{EARLY}$, $guard_{POD}$, $guard_{EARLY}$ and $guard_{LATE}$ defined earlier in Section III-C2 and used in Figure 10. Note the absence of a parameter for $event_{LATE}$. The third parameter in line 1, *init_rt*, contains the initial stoplight state. The role of the last two parameters, *xPOD* and *xEARLY*, is to reduce the number of *rt* instances needed to code constraints. Parameter *xPOD*, when TRUE, indicates that both rising and falling signal transitions count as events for *myPOD*. Parameter *xEARLY* indicates the same for *myEARLY*. The last two parameters make it possible to model *inv_q2d.val*± in *p7* and *p8* of Figure 12 with a single *rt* instance by making *xEARLY* TRUE (t) — see line 27.

The statement between the keywords *case* and *esac* in lines 7 and 14 is precisely the code for the stoplight's state changes. It is executed in synchronous mode, i.e. in each execution step by the model checker, as indicated by the keyword *TRANS* in line 6. This is consistent with the mode of execution indicated earlier in Figure 4 for the leftmost white rectangle with the name "Component's Timing Constraints."

The signal, *stop*, defined in line 18 of Figure 13, is TRUE if and and only if the stoplight is RED.

The constraint's *rt* instances follow in lines 22–30, in the code for the *circuit* module. Constraints that start in the same state and that use the same *myPOD*, *myEARLY*, and $guard_{LATE}$

have the same *rt* parameters in the NuSMV code, and can thus share an *rt* instance. For instance, patterns *p1* and *p3* share an *rt* instance in line 24, called *p1p3*. This is possible, despite the fact that the two patterns have different *myLATE* events. It is possible because the *rt* instances control the GREEN, YELLOW and RED stoplight states, but they do not block *myLATE* event. The *cgate* instance that drives the *myLATE* event, event_{LATE}, is the one that blocks the event.

For the model checker, we partitioned the stoplight model of Figure 10 (left) into a stoplight controller (*rt*) and a driver (*cgate*). Although [37] uses a two-state model instead of our three-state stoplight model, their model checker solution uses exactly the same partition. The analogy with everyday stoplights and drivers makes this an obvious partition.

Just as multiple stoplights may force a driver to stop a vehicle, so multiple *rt* instances may force a *cgate* to block a *myLATE* event. Take for instance relative timing constraints *p1*, *p4*, *p5*, and *p7* of Figure 12. The constraints share *myLATE* event, *in1_R*±, and may thus each block it. Thus, in the model checker, each of the *stop* signals of the constraints' multiple *rt* instances may block *in1_R*±. Line 29 of Figure 13 combines these separate *stop* signals into a single variable *stop_in1_R_x*, to simplify the remaining code. Likewise, line 30 combines the separate *stop* signals for *myLATE* event *out1_A*± into a single variable *stop_out1_A_x*.

All events *in1_R*± and *out1_A*± are generated by the environment. Therefore, *stop_in1_R_x* and *stop_out1_A_x* must pass from the instantiated circuit to the instantiated environment through the parameter mechanism, as shown in lines 41–48 of Figure 13.[11] In lines 34–37, module *environment* passes the parameters to the appropriate *cgate* drivers, *ENV_in1* and *ENV_out1*. Because *stop_in1_R_x* blocks the rising as well as the falling transitions of *ENV_in1.val* it is passed to both the *stop_rise* and the *stop_fall* parameter slots for *cgate ENV_in1* — making the *cgate* block any transition of *ENV_in1.val* if *stop_in1_R_x* is TRUE (line 9 of Figure 7). This is how timing constraints control the run-time values of *stop_rise* and *stop_fall* in the various *cgate* drivers of *myLATE* events.

It is straightforward to generate the NuSMV code changes in lines 21–49 of Figure 13 from the timing patterns in Figure 12. Module *rt*, in lines 1–18 of Figure 13, is a predefined module in ARCtimer's Model Checker Library for NuSMV, just like the modules for combinational logic gate *cgate* and for positive edge-triggered flipflop *ff_posedge*.

III-E ARCtimer Step 4 — Static Timing Analysis

Section III-C3 ended Step 3 "Timing Patterns" of Figure 1 (right-column-bottom) with a set of timing patterns *p1* to *p8* and their parametrized version *p* — see Figure 12. This set is complete in terms of property coverage and sufficiently general to apply to every chip with a Click Storage component.

This Section III-E takes *p* to Step 4 "Static Timing Analysis" of Figure 1 (center-column-bottom) by translating *p*'s formula into code for static timing analysis (STA). We store both *p*'s formula, $p : \mathit{myPOD} \rightarrow \mathit{myEARLY} < \mathit{myLATE}$, and its STA code in the Design Library.[12]

The first task for static timing analysis is to validate *p*, i.e. to validate that *p*'s slowest early path is faster than *p*'s fastest late path in the chip's gate-level netlist. This involves computing the maximum path delay, max_{EARLY}, of all paths from *myPOD* to *myEARLY* and the minimum path delay, min_{LATE}, from *myPOD* to *myLATE*, and validating that:

- $\mathit{max}_{EARLY} < (\mathit{min}_{LATE} + \mathit{margin})$
  for some delay *margin*.

The second task for static timing analysis is to repair the netlist in case the first task invalidates *p*. The iterative repair process described in Section II-D performs this second task. It finds the minimum delay value d to make *p* valid, given a delay insertion point in the netlist at which to insert d.

Calculating max_{EARLY} and min_{LATE} involves following the topological connections between gates and wires in the netlist, differentiating rising from falling transitions where possible, and filling in gate and wire delays using lookup tables [49], [50], [22], [23], [51], [25]. Unfortunately, some STA tools cannot differentiate rising from falling transitions, and many STA tools cut paths and loops at flipflops — see Section II-D. As a result, most STA tools need guidance to know if a path passes through or bypasses a flipflop and to know which delay to use at asymmetric delay insertion points.

Various self-timed design groups have developed solutions to guide STA tools through a gate-level netlist with self-timed circuitry — see Section II-D. The solutions usually involve pre-cut sub-paths that a conventional STA tool can handle. These pre-cut sub-paths are the result of a higher-level analysis of the netlist. The higher-level analysis is the most interesting part of any of these solutions, because it is the part that would remain necessary even if conventional STA tools were capable of doing the analysis without guidance.

The STA code stored in the Design Library does the higher-level analysis. It contains the algorithms to find paths and calculate path delays and to mark intermediary flipflops and other relevant checkpoints on the paths.[13] Below, we indicate the most important decisions that we made to organize this STA code. These decisions complement the actual path cutting pragmatics described for instance in [50], [22], [51], [25].

- *Fill in crucial semantic details in advance:*
  We use the model checker and formal analysis to fill in behavioral details that a topological search cannot find.
- *Mimic the modularity of the self-timed design:*
  Because the Design Library stores information by component, we must partition the STA code also by component.

[11] Some timing constraints exchange parameters in the reverse direction, from the environment to the circuit. For example, bundled data setup time constraints pass *myPOD* from the environment to the circuit — see footnote 3.

[12] The STA code for non-relative-timing constraints, such as minimum clock pulse widths, can be stored, organized, and generated in a way similar to *p*.

[13] We call these *checkpoints*, after the Berlin Wall's "Checkpoint Charlie" — the famous Cold War crossing point between East and West Berlin.

  Our self-timed components communicate by handshakes over channels, as explained in Section III-A and Figure 2. Each handshake is marked by a pair of events, making the channel full and then empty, or vice versa. In Click these events are marked by a transition on the request signal followed by a transition on the acknowledge signal, or vice versa. Each pair of handshake events partitions the paths in the netlist between two successive components.

  Thus, although we store the main STA code for validating the component's timing constraints with the component, we can distribute the full code by storing the delay calculations for the other side of a partitioned path with the other component.[14]

  The STA code uses the pair of handshake events to initiate an external delay calculation and return its results. This process may be recursive, because the STA code for the neighboring component may initiate sub-calculations stored with further out neighboring components before it can complete its calculation.

  We implement this using *channel subroutine calls*. Hence, in addition to STA code for validating its own timing constraint, each component must also store STA code for the *channel subroutines* for which it might receive calls.

- *Sequence the calculations in a sensible way*
  Internal paths generally contribute less delay than paths that exit and enter the component via a handshake channel. Thus, it makes sense to start minimum path delay calculations with internal paths, and use the current minimum to cut off subsequent calculations including the channel subroutine calls introduced above.

As an example, let us look at the decisions and STA code organization related to timing constraint *p* of Figure 12 and its calculations to validate $max_{EARLY} < (min_{LATE} + margin)$:

- *Delay insertion points:*
  We have chosen to repair *p* at the two *myLATE* events, by delaying signal changes on *in[n2]_R* and *out[m2]_A* — whichever applies. The two end signals make good repair points, because, not only do they change exactly once per *myPOD–myLATE* cycle, the minimum frequency for repair, but also their change covers all of the *myEARLY* events in each *myPOD–myLATE* cycle. The delay element must delay both rising and falling transitions because the direction of the change is irrelevant, as indicated by the symbol "±" in Figure 12.

  Also, as *p*'s *myLATE* events, *in[n2]_R* and *out[m2]_A* share the same set of *myEARLY* events. As a result, we can delay signal changes on *in[n2]_R* and *out[m2]_A* without creating circular repair dependencies. The lack of circular dependencies ensures that the repair process described in Section II-D will converge.

- *Additional semantic details to simplify* max_{EARLY}*:*
  Timing constraint *p* has falling signal transitions for *myEARLY* events, and thus requires transition-aware static timing analysis. However, the only *myEARLY* event preventing a transition-agnostic analysis is *buf_ck.val*−. Unlike its transition-agnostic version *buf_ck.val*±, event *buf_ck.val*− follows *and2.val*+ not immediately but only after *FF.q*±. As it happens, all *p*'s *myEARLY* events follow *and2.val*+ after *FF.q*±. We can indicate this by adding *FF.q*± between *myPOD* and *myEARLY* in *p*. The presence of *FF.q*± makes it possible to focus on changes rather than specific transitions of *myEARLY* events — the transitions are implied, as the model checker can confirm. Figure 14 shows the updated version of *p*, with checkpoint *FF.q*± and non-specific *myEARLY* event transitions. This is the version that we translate into STA code, using transition-agnostic calculations.[15]

  From *p* itself we can deduce that early paths end before any *myLATE* event and thus never go through a channel. Therefore, we can use transition-agnostic calculations for max_{EARLY} restricted to paths internal to the module.

- *STA code partitioning for* min_{LATE}*:*
  The min_{LATE} STA code for p calculates the minimum path delay for paths from *myPOD* to *myLATE* internal to the module — no holds barred. The code keeps track of any flipflop or other checkpoints that may need further preparation before the calculations can be handed over to conventional STA tools. Each time the path subsequently exits and enters the module over a channel, the code inserts a channel subroutine call, and splits the calculation into the sum of three sub-calculations: the original calculation up to the exit over the channel, the channel subroutine call, and the original calculation from the entry over the channel back into the module.

  Each channel comes with STA code to fill in the delay of a channel subroutine call entering and exiting the Click Storage module. Each such channel subroutine calculates the minimum path delay for paths through the component from channel entry to channel exit — no holds barred.

Our STA code calculations are conservative. They tend to ignore any guards, and focus only on the event changes in the relative timing constraint formulation of Section III-C2. This is possible because we compensate for missing details by adding checkpoints. Moreover, static timing validation is more forgiving than behavior-based timing verification: it suffices

[14] Bundled data setup and hold time constraints use a similar partition based on different pairs of handshake events — between request and data signals versus acknowledge and data signals. See also footnote 3.

[15] We chose *and2.val*+ as *p*'s *myPOD* rather than *FF.q*±, because as the AND function of the Click Storage component, *and2.val*+ makes the component "act" more so than *FF.q*±. Moreover, alternative circuit implementations that split *FF* into separate flipflops for each channel [52] require *p* to use *and2.val*+ as *myPOD*. Having said that, the Click Storage circuit in Figure 1 (right-column-top) can use *FF.q*± as *myPOD* and thus, without inserting an additional checkpoint, avoid the need to differentiate rising from falling transitions in *myEARLY*. Whether one chooses to keep the STA code as general as possible by taking *and2.val*+ as *myPOD* or as simple as possible by taking *FF.q*± as *myPOD* the goal remains the same: simplify the STA code using transition-agnostic path-finding and path-delay calculations where possible.

to satisfy $max_{EARLY} < (min_{LATE} + margin)$ even were the minimum and maximum delays to belong to false paths.

Note:
The signal names that we obtain from the model checker all specify gate outputs — never gate inputs. However, *p*'s *myEARLY* and *myLATE* implicitly represent events that happen not just at the output of the gate but also in the wires branching out to each subsequent gate's input. We must code these events accordingly lest there be gaps in the coverage of *p*. The pseudo-random delay insertion test scenario described at the end of Section II-D might or might not detect such coverage gaps. Therefore, instead of looking for changes at *p*'s *myEARLY* and *myLATE* output signals, the STA code looks for changes at the gate inputs connected to these signals. For example, instead of looking for *buf_ck.val*±, a change at the clock buffer output in Figure 1 (right-column-top) the STA code looks for *FF.ck*±, a change at the flipflop's clock input.

III-F Summary Timing Verification Framework Steps 1–4

Our Design Library, Figure 1 (center-column), stores a set of handshake components for use in larger self-timed systems. For each component, the Design Library stores a circuit description, a protocol description, a description of the timing constraints for the circuit, and static timing analysis code to validate and enforce these constraints in the final system. The circuit and its timing constraints are known to follow the protocol properly because they have gone through the verification steps outlined here. Because these verification steps happen early in the design process, we have the leisure to apply an in-depth verification process.

We make use of a model checker as part of the verification process. The model checker verifies that each component, or rather its timing constrained circuit, obeys the protocol specified for its interface signals. The model checker also verifies the "digital health" of a component. "Digital health" includes such properties as *semimodular gate behavior* and as *absence of set-reset drive fights*, a "digital health" property not used in this paper but important for verification of GasP components. The static timing analysis code covers additional timing constraints, such as minimum clock pulse widths for all the edge-triggered flipflops in the Click circuit. The flipflop models in this paper are too abstract for the model checker to detect the need for such pulse width constraints.

In building the Design Library, we strive for modularity and generality. The Design Library is organized by component. Even the static timing analysis code generated in Step 4 is partitioned over the components. For each handshake component we seek circuit descriptions as well as protocol, constraint, and code descriptions that are understandable to the component's designer, easy to maintain, and robust to circuit modifications applied later in the design process. Where possible, the descriptions in the Design Library are parametrized to address variable numbers of channels. Whenever we use the term *pattern*, as in *design pattern* or *timing pattern*, it is to emphasize the generality of that particular description.

IV Comparison to Related Work

Throughout this paper, we have identified related work in context. We have identified essential decisions and explained where, how, and why our decisions differ from those of others. For instance, in Section II-D we highlighted three static timing analysis (STA) decisions that one must make: where and when to insert delay to repair invalid timing constraints, and what STA engine to use. The purpose of this Section IV is to highlight where our key decisions are new or different.

One important choice was to select a general-purpose model checker rather than one specialized to the timing verification of self-timed circuits. The active and diverse user community of the NuSMV model checker that we use in this paper gave us high confidence that the software would be correct as well as enable us to control our experiments. The work reported in [37] also uses the NuSMV general-purpose model checker. Even so, model checkers developed specifically for timing verification of self-timed circuits — notably [33], [26] — have important value because of the new theories they founded and the new experiments they enabled. However, our experience with specialized model checkers for tool support has been a struggle due to hidden assumptions and hard to find bugs, possibly reinforced by the monolithic solution approach.

We model semimodular gate behavior and generate timing constraints that guarantee a gate's semimodular behavior if needed, as do [33], [29], [26], [37]. But we use a new definition of semimodularity, adapted for relative timing, which we first published in [42]. The new definition, coded in lines 14–18 of Figure 7, prevents blocked gate output transitions from causing semimodularity failures. The idea is simple: a transition that is blocked by a timing constraint is already disabled and therefore cannot be disabled further. Note that the general gate models that we use and share with [43] make it much easier to swap in a new definition of semimodularity than it would be were we to use a dedicated model for each gate with a different logic function as is done in [23], [37].

Like [33], [29], [23], [37] we verify that a component, or rather its timing constrained circuit, obeys the protocol specified for its interface signals. This includes verifying that the component's handshake transitions are legal, as expressed by the safety properties in lines 30–31 of Figure 5. In our work and that of [33], obeying the protocol also means that the circuit can make progress in certain states even if the environment fails to make progress, as expressed in lines 33–37 of Figure 5. Progress under a lazy environment is modeled in the theory of Delay-Insensitive Algebra, but not in the theory underlying [29], [23], [37]. Unlike others, we also verify that all reachable protocol scenarios remain available for implementation. This is expressed by the choice equivalence properties coded in lines 39–50 of Figure 5. Choice equivalence properties are *meta properties* to check whether or not the verification process overlooked a protocol scenario. They target not an individual timing constrained circuit but the model checker's finite state machine model of all possible timing constrained circuits.

The timing constraints that we model and verify constrain the ordering of specific events that originate from a common start event. We formulate and name these constraints in a way similar to [23], [37], by specifying the common start event, and the constrained early and late events. The resulting event ordering specifications are called *relative timing constraints*, after [41]. Because of their simple formulation, relative timing constraints are easy to model and add to an already existing netlist configuration in the model checker, as illustrated in Section III-C2 and Section III-D. But the same formulations that are so simple to read, model, and verify can be difficult to translate for static timing analysis (STA). Translation into STA code may require one to specify intermediary checkpoints and corresponding events, as illustrated in Section III-E and Figure 14. The addition of checkpoint events makes a relative timing constraint look more like a *chain constraint* [33], [35]. Chain constraints specify the paths over which the delays to the early and late events must be calculated. As such, chain constraints already contain most of the semantic details that we add in Section III-E to simplify STA code generation. Consequently, chain constraints are much easier to translate into STA code than relative timing constraints. However, chain constraints are also much harder to model than relative timing constraints.

Fortunately, we seldom need to add more than a few checkpoints to simplify STA code generation. These checkpoints can be added systematically to existing relative timing constraints, using the *stoplight* model of Figure 10 (left), as demonstrated in Figure 14. This 3-state model is new. It generalizes the 2-state model in [37] to guarded events — the guard indicates whether or not the event instance applies.

The key focus in this paper, more so than in any related work seen, is to ensure that not only the circuit of a handshake component but also its timing constraints and static timing analysis code are sufficiently general for use in a Design Library. This paper is silent about using timing constraints to guide synthesis and layout of the design or parts of it, as explored in for instance [23], [53], [25], but we anticipate that such extensions will follow a similar pattern.

V Summary and Conclusion

This paper introduces ARCtimer, a framework for generating and verifying timing constraints for handshake components, as needed to make the component's gate-level circuit follow the component's handshake protocol. The component thus verified goes into a library for later system use. This library, the Design Library of Figure 1 (center-column), stores general descriptions, called patterns, of the component's circuit and protocol and timing constraints. The Design Library also stores static timing analysis code to validate and enforce the component's constraints in any self-timed system built using the library. Because the timing constraints ensure that each component faithfully follows its protocol, and because each protocol is delay insensitive, the resulting systems are delay insensitive. *By constraining time locally, we free it globally* — this hallmark of self-timed design established since Seitz' *equipotential regions* and *self-timed signaling* [54] applies not only to designing and parsing self-timed systems but also to verifying and validating them, as this paper confirms.

We wrote this paper to help readers understand tradeoffs and decisions. It explains where in the design flow a framework like ARCtimer fits. It identifies essential decision points, the choices one can make, what we and others chose, and why. Although this paper is not intended as a survey, it refers to many related works and discusses them in context. We encourage readers to reproduce our work and our results. To enable readers to do so, the paper provides high-level algorithmic descriptions where possible, as well as low-level details where we think it helps to have a starter set of code that is known to work. We show our NuSMV model checker code not only to give readers a head start, but also to indicate that it is not hard to generate such code automatically from the component information stored in the Design Library.

Last but not least: we call upon readers for collaboration in moving this field forward. Many of today's self-timed design tools are monolithic. The time has come to make exchangeable theory and tool parts. This paper identifies three problem areas shared by all self-timed circuit families and thus prime areas for exchangeable solutions. The three problem areas are: static timing analysis with loops kept intact (Section II-D), delay-insensitive protocol specifications (Section III-A), and failure analysis heuristics to derive timing constraints (Section III-C, footnote 10). This paper also identifies a solution approach to facilitate the exchange of theory and tools between self-timed circuit families by using the organizational discipline that binds them all: *design patterns*.

Acknowledgment

We thank private and corporate sponsors of the Asynchronous Research Center (ARC) at Portland State University — without their support this work would not be possible. We are fortunate to have the support and encouragement of our Deans, Dean Renjeng Su at Portland State University and Dean Bin Hu at Lanzhou University. Special thanks go to Willem Mallon for the world-class compiler skills, experience in self-timed design, and wisdom he brought to the ARC. Willem not only built ARCwelder, a design compiler for self-timed systems whose development shaped our work, but also brought Delay-Insensitive Algebra to our attention. We owe much to his contributions. We thank Anping He's student at Lanzhou University, Chao Zhou, for automating the protocol translation for the NuSMV model checker. We thank also ARC students Chris Cowan, Navaneeth Jamadagni, and Swetha Mettala Gilla for daily companionship, discussion, and laughter.

References

[1] Jun Sawada and Warren A. Hunt Jr. Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability. *Formal Methods in System Design*, 20(2):187–222, 2002.

[2] Anna Slobodová, Jared Davis, Sol Swords, and Warren Hunt Jr. A Flexible Formal Verification Framework for Industrial Scale Validation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 89–97, 2011.

[3] Chao Yan, Florent Ouchet, Laurent Fesquet, and Katell Morin-Allory. Formal Verification of C-element Circuits. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 55–64, 2011.

[4] Ivan Sutherland, Bob Sproull, and David Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, 1999.

[5] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

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

[7] Doug Edwards and Andrew Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[8] Peter A. Beerel, Recep O. Ozdag, and Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[9] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 46–53, 2001.

[10] Ivan Sutherland. GasP Circuits that Work. ECE 507 Research Seminar, Fall 2010. Asynchronous Research Center, Portland State University. Download from: http://arc.cecs.pdx.edu/fall10.

[11] Alain J. Martin and Mika Nyström. Asynchronous Techniques for System-on-Chip Design. *Proceedings of the IEEE*, 94(6):1089–1120, 2006.

[12] Basit Riaz Sheikh and Rajit Manohar. "energy-efficient pipeline templates for high-performance asynchronous circuits". *ACM Journal on Emerging Technologies in Computing Systems*, 7(4), 2011.

[13] Montek Singh and Steven M. Nowick. "mousetrap: High-speed transition-signaling asynchronous pipelines". *IEEE Transactions on Very Large Integration (VLSI) Systems*, 15(6):684–698, 2007.

[14] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 3–14, 2010.

[15] Louis Scheffer, Luciano Lavagno, and Grant Martin (Eds.). *Electronic Design Automation for Integrated Circuits Handbook, Volumes 1–2*. CRC Press and Taylor & Francis, 2006.

[16] Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalij, and Ad Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, 1994.

[17] Joep Kessels and Paul Marston. Designing Asynchronous Standby Circuits for a Low-Power Pager. *Proceedings of the IEEE*, 87(2):257–267, 1999.

[18] Kenneth S. Stevens, Shai Rotem, Ran Ginosar, Peter Beerel, Chris J. Myers, Kenneth Y. Yun, Rakefet Kol, Charles Dike, and Marly Roncken. An Asynchronous Instruction Length Decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, 2001.

[19] Basit Riaz Sheikh and Rajit Manohar. An Operand-Optimized Asynchronous IEEE 754 Double-Precision FLoating-Point Adder. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 151–162, 2010.

[20] Basit Riaz Sheikh and Rajit Manohar. An Asynchronous FLoating-Point Multiplier. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 89–96, 2012.

[21] Sachin S. Sapatnekar. Chapter 6: Static Timing Analysis. In *L. Scheffer, L. Lavagno, G. Martin (Eds.): Electronic Design Automation for Integrated Circuits Handbook, Volume 2*. CRC Press and Taylor & Francis, 2006.

[22] Mallika Prakash and Peter A. Beerel. Static Timing Analysis of Template-Based Asynchronous Circuits. *US Patent US 2009/0210841 A1*, assigned to the University of Southern California, August 20, 2009.

[23] Kenneth S. Stevens, Yang Xu, and Vikas Vij. Characterization of Asynchronous Templates for Integration into Clocked CAD Flows. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 151–161, 2009.

[24] Peter A. Beerel, Georgios D. Dimou, and Andrew M. Lines. Proteus: An ASIC Flow for GHz Asynchronous Designs. *IEEE Design & Test of Computers*, 28(5):36–51, 2011.

[25] Vikas S. Vij. *Algorithms and Methodology to Design Asynchronous Circuits Using Synchronous CAD Tools and Flows*. PhD thesis, Electrical and Computer Engineering, The University of Utah, USA, 2013.

[26] Yang Xu. *Algorithms for Automatic Generation of Relative Timing Constraints*. PhD thesis, Electrical and Computer Engineering, The University of Utah, USA, 2011.

[27] Teresa Meng. *Synchronization Design for Digital Systems*. Kluwer Academic Publishers, 1991.

[28] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Kluwer Academic Publishers, 1993.

[29] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.

[30] Mark B. Josephs and Jan Tijmen Udding. An Algebra for Delay-Insensitive Circuits. In *Computer Aided Verification (CAV)*, pages 343–352, 1991.

[31] Tom Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1994.

[32] Willem C. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. PhD thesis, Mathematics and Natural Sciences, University of Groningen, The Netherlands, 2000.

[33] Radu Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. PhD thesis, Computer Science, University of Waterloo, Canada, 1998.

[34] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[35] Radu Negulescu and Ad Peeters. Verification of Speed-Dependences in Single-Rail Handshake Circuits. In *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 159–170, 1998.

[36] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. NuSMV 2.4 User Manual, 2013. Download from http://nusmv.fbk.eu/NuSMV/userman/index-v2.html.

[37] Krishnaji Desai, Kenneth S. Stevens, and John O'Leary. Symbolic Verification of Timed Asynchronous Hardware Protocols. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 147–152, 2013.

[38] David E. Muller and W. Scott Bartky. A Theory of Asynchronous Circuits. In *International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, 1959.

[39] Raymond E. Miller. *Switching Theory Volume 2: Sequential Circuits and Machines,* (Chapters 9–10: Asynchronous Switching Networks & Speed Independent Switching Circuit Theory). John Wiley & Sons, 1965.

[40] Peter A. Beerel and Marly E. Roncken. Low Power and Energy Efficient Asynchronous Design. *Journal of Low Power Electronics (JOLPE)*, 3(3):234–253, 2007.

[41] Kenneth S. Stevens, Ran Ginosar, and Shai Rotem. Relative Timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, 2003.

[42] Hoon Park, Anping He, Marly Roncken, and Xiaoyu Song. Semi-Modular delay model revisited in context of relative timing. *Electronics Letters*, 51(4):332–334, 2015.

[43] Khaled Alsayeg, Katell Morin-Allory, and Laurent Fesquet. RAT-based formal verification of QDI asynchronous controllers. In *Forum on Specification & Design Languages (FDL)*, pages 1–6, 2009.

[44] Tomohiro Yoneda, Tomoya Kitai, and Chris Myers. Automatic Derivation of Timing Constraints by Failure Analysis. In *Computer Aided Verification (CAV)*, pages 195–208, 2002.

[45] Marco A. Peña, Jordi Cortadella, Alex Kondratyev, and Enric Pastor. Formal verification of safety properties in timed circuits. In *IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 2–11, 2000.

[46] Hoshik Kim, Peter A. Beerel, and Ken Stevens. Relative Timing Based Verification of Timed Circuits and Systems. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 115–124, 2002.

[47] Yang Xu and Kenneth S. Stevens. Automatic Synthesis of Computation Interference Constraints for Relative Timing Verification. In *IEEE*

International Conference on Computer Design (ICCD), pages 16–22, 2009.

[48] Robert M. Fuhrer and Steven M. Nowick. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools*. Kluwer Academic Publishers, 2001.

[49] Mallika Prakash. Library Characterization and Static Timing Analysis of Asynchronous Circuits. Master's thesis, Computer Engineering, University of Southern California, USA, December 2007.

[50] Prasad Joshi. Static Timing Analysis of GasP. Master's thesis, Electrical Engineering, University of Southern California, USA, December 2008.

[51] Swetha Mettala Gilla. Library Characterization and Static Timing Analysis of Single-Track Circuits in GasP. Master's thesis, Electrical and Computer Engineering, Portland State University, USA, 2010.

[52] Marly Roncken, Swetha Mettala Gilla, Hoon Park, and Ivan Sutherland. Naturalized Communication and Testing. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2015.

[53] Minoru Iizuka, Naohiro Hamada, Hiroshi Saito, Ryoichi Yamaguchi, and Minoru Yoshinaga. A Tool Set for the Design of Asynchronous Circuits with Bundled-data Implementation. In *IEEE International Conference on Computer Design (ICCD)*, pages 78–83, 2011.

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

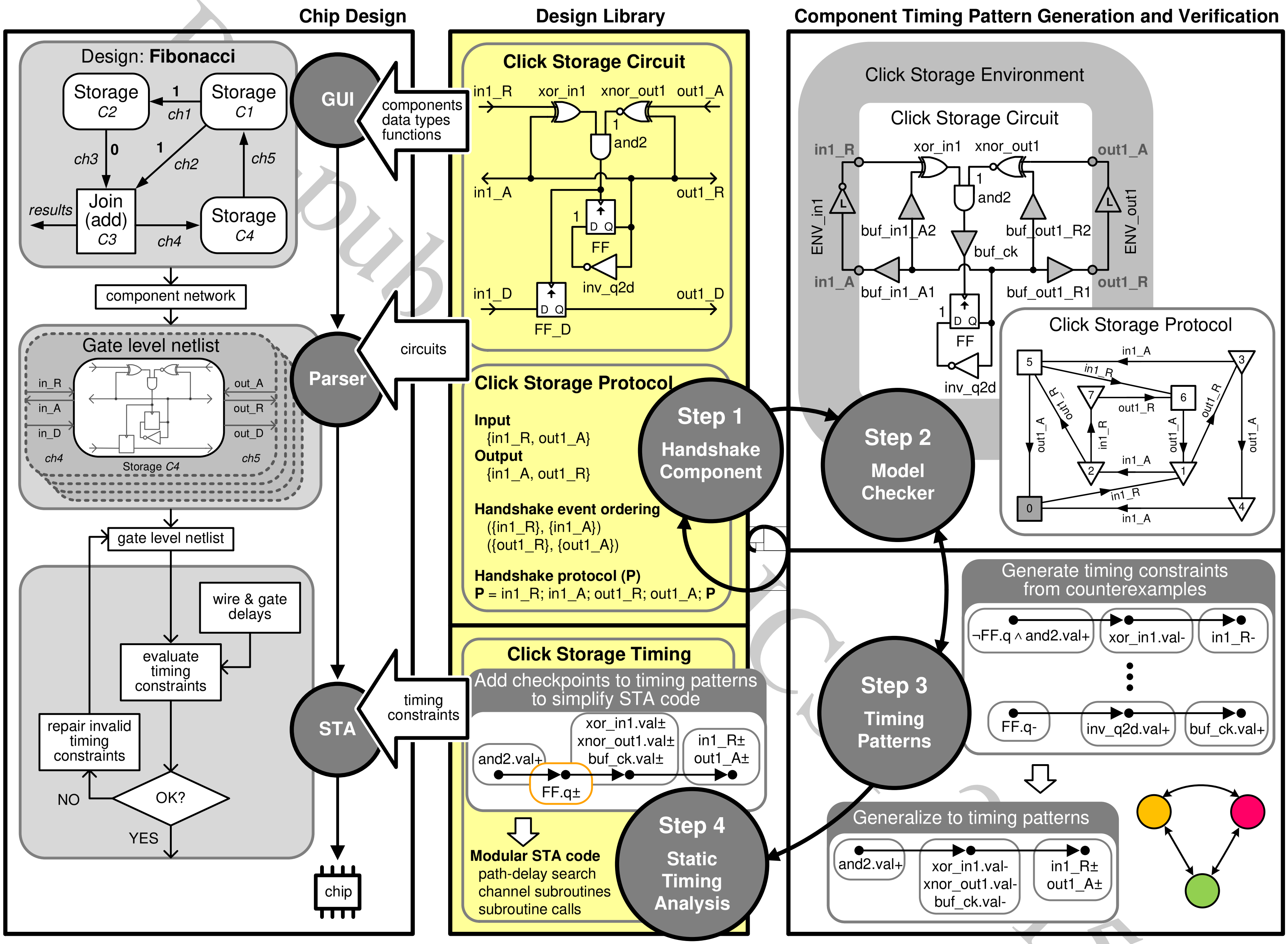


Figure 1 Reference diagram for this paper, illustrating the timing verification context and framework of ARCtimer. The *Design Library* in the center column connects the *Chip Design* flow on the left and the *Component Timing Pattern Generation and Verification* framework (ARCtimer) on the right. Once verified, a component may be used in many chip designs. The library stores the component's circuit (center-top), protocol (center-middle), its timing constraints, and the static timing analysis code to enforce these constraints in the final chip design (center-bottom). The chip design flow and library are explained in Section II. ARCtimer Step 1 through Step 4 follow in Section III.

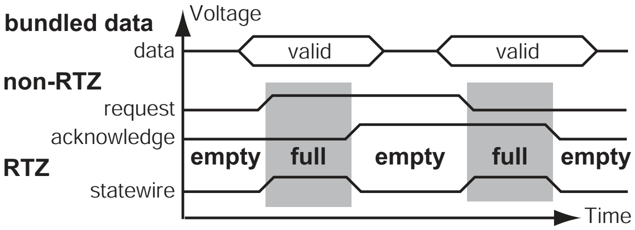


Figure 2 Default state representations for full and empty channels in two-phase non-RTZ and RTZ handshake protocols with bundled data. A channel with non-RTZ protocol is engaged in a handshake, i.e. is full, when its *request* and *acknowledge* differ. A channel with RTZ protocol is full when its *statewire* is high. During the handshake, i.e. when the channel is full, data must be valid and remain stable.

Input:
{in1_R , out1_A}

Output:
{in1_A , out1_R}

Handshake event ordering:
({in1_R} , {in1_A})
({out1_R} , {out1_A})

Handshake protocol (P):
P = in1_R ; in1_A ; out1_R ; out1_A ; P

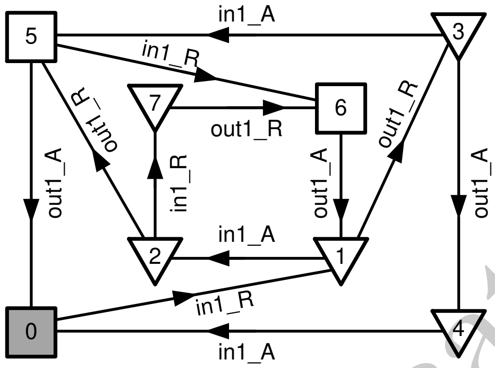


Figure 3 (From Figure 1 — center and right columns) Compact (top) and complete (bottom) protocol specification of a Click Storage component with a single incoming and a single outgoing handshake channel. The finite state machine (bottom) starts in state 0, colored grey. The triangles (▽) indicate a transient state. No matter the environment that it operates in, the component must exit a transient state. It can exit a transient state through an event on one of the state's outgoing arrows. The arrow points to the component's next state. The rectangles (□) indicate a non-transient state. In some environments, the component will stay in such a state forever; in other environments, it exits the state by following one of the state's event-arrow pairs to the next state.

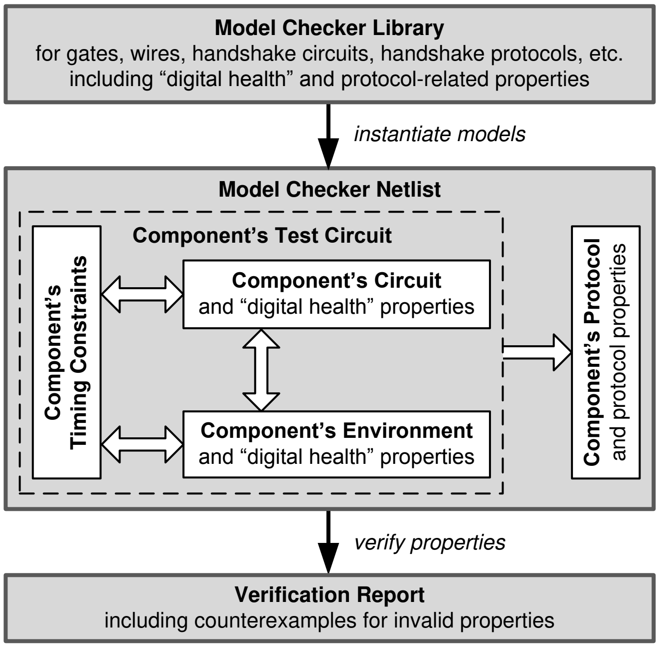
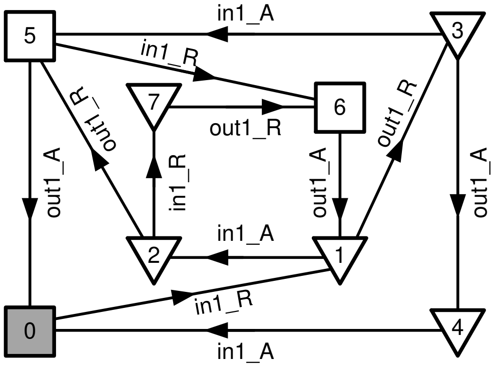
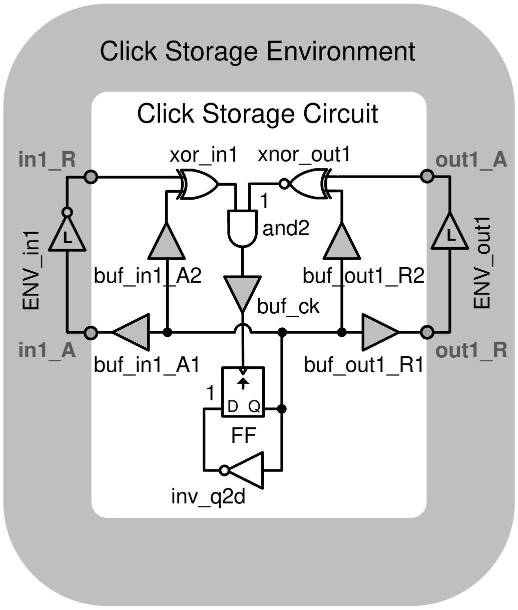

Model Checker Library
for gates, wires, handshake circuits, handshake protocols, etc.
including "digital health" and protocol-related properties
instantiate models
Model Checker Netlist
Component's Test Circuit
Component's Timing Constraints
Component's Circuit
and "digital health" properties
Component's Environment
and "digital health" properties
Component's Protocol
and protocol properties
verify properties
Verification Report
including counterexamples for invalid properties

Figure 4 Organization of the model checking task to verify, for a given handshake component, that the component's circuit in its environment and under its timing constraints satisfies both the gate-level "digital health" properties and the properties defined by the component's protocol. Examples of "digital health" are semimodularity, used later in this paper, and absence of set-reset drive fights, which plays no role in this paper. The grey rectangle at the top represents the *Model Checker Library* — a translation of the Design Library in Figure 1 (center-column) into model checker lingo. Using the Model Checker Library, ARCtimer creates a *Model Checker Netlist* represented by the middle grey rectangle. The netlist connects single instances of the component's protocol, circuit, environment, and available timing constraints, as indicated by the white and broken-line rectangles and the white arrows. A white arrow follows the direction from a rectangle with models that create an event to a rectangle with models that respond to that event. All events in and between *rectangles with horizontal text* are interleaved using an asynchronous mode of operation. Events of *rectangles with vertical text* must be synchronized to corresponding events, which is achieved by operating them in synchronous mode. The model checker takes the Model Checker Netlist and first generates a corresponding finite state machine model with instantiated gate-level "digital health" and protocol-related properties for verification, and then it checks the properties. The grey rectangle at the bottom represents the verification report with a pass or fail indication per property and a counterexample of a computation path in the resulting finite state machine for each failing property.



```
1   MODULE protocol (in1_R, in1_A, out1_R, out1_A)
    VAR
      state: {s0, s1, s2, s3, s4, s5, s6, s7, errorOUT, errorIN};
    ASSIGN
5     init(state) := s0;
    TRANS
      next(state) = case
        --legal handshake transitions
        state=s0  & (in1_R  != next(in1_R))   : s1;
10      state=s1  & (in1_A  != next(in1_A))   : s2;
        state=s1  & (out1_R != next(out1_R))  : s3;
        state=s2  & (in1_R  != next(in1_R))   : s7;
        state=s2  & (out1_R != next(out1_R))  : s5;
        state=s3  & (out1_A != next(out1_A))  : s4;
15      state=s3  & (in1_A  != next(in1_A))   : s5;
        state=s4  & (in1_A  != next(in1_A))   : s0;
        state=s5  & (in1_R  != next(in1_R))   : s6;
        state=s5  & (out1_A != next(out1_A))  : s0;
        state=s6  & (out1_A != next(out1_A))  : s1;
20      state=s7  & (out1_R != next(out1_R))  : s6;
        --illegal handshake transitions
        in1_A!=next(in1_A) | out1_R!=next(out1_R) : errorOUT;
        in1_R!=next(in1_R) | out1_A!=next(out1_A) : errorIN;
        --remaining transitions
25      TRUE: state;
      esac;

    --PROPERTIES
      --safety
30      CTLSPEC AG state!=errorOUT;
        CTLSPEC AG state!=errorIN;
      --progress
        CTLSPEC AG AF(state!=s1)
        CTLSPEC AG AF(state!=s2)
35      CTLSPEC AG AF(state!=s3)
        CTLSPEC AG AF(state!=s4)
        CTLSPEC AG AF(state!=s7)
      --choice equivalence
        CTLSPEC AG (state=s0 → E[state=s0 U state=s1])
40      CTLSPEC AG (state=s1 → E[state=s1 U state=s2])
        CTLSPEC AG (state=s1 → E[state=s1 U state=s3])
        CTLSPEC AG (state=s2 → E[state=s2 U state=s7])
        CTLSPEC AG (state=s2 → E[state=s2 U state=s5])
        CTLSPEC AG (state=s3 → E[state=s3 U state=s4])
45      CTLSPEC AG (state=s3 → E[state=s3 U state=s5])
        CTLSPEC AG (state=s4 → E[state=s4 U state=s0])
        CTLSPEC AG (state=s5 → E[state=s5 U state=s6])
        CTLSPEC AG (state=s5 → E[state=s5 U state=s0])
        CTLSPEC AG (state=s6 → E[state=s6 U state=s1])
50      CTLSPEC AG (state=s7 → E[state=s7 U state=s6])
```

Figure 5 Fully delay-insensitive protocol specification (top) and corresponding NuSMV model checker code (bottom). The first part of the code in lines 1–26 describes legal and illegal states and transitions. The model checker uses this part to monitor the sub-system with circuit, environment, and timing constraints. The second part in lines 28–50 describes the protocol properties for "what the monitor must see." **Note:** A NuSMV case statement gives higher priority to the guarded commands in earlier lines of the case statement.

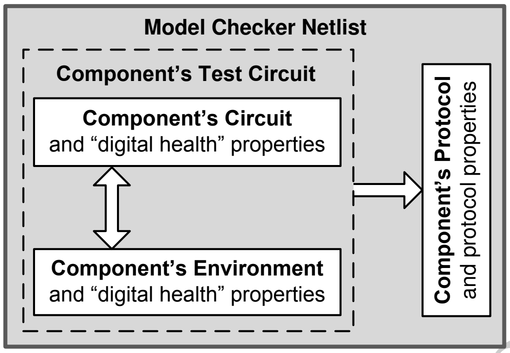


```
MODULE circuit (in1_R, out1_A)
VAR
  xor_in1      : process cgate (in1_R xor buf_in1_A2.val, f,f,f,f);
  xnor_out1    : process cgate (out1_A xnor buf_out1_R2.val, t,f,f,f);
  and2         : process cgate (xor_in1.val & xnor_out1.val, f,f,f,f);
  buf_ck       : process cgate (and2.val, f,f,f,f);
  FF           : ff_posedge (buf_ck.val, inv_q2d.val, f);
  inv_q2d      : process cgate (!FF.q, t,f,f,f);
  buf_in1_A1   : process cgate (FF.q, f,f,f,f);
  buf_in1_A2   : process cgate (FF.q, f,f,f,f);
  buf_out1_R1: process cgate (FF.q, f,f,f,f);
  buf_out1_R2: process cgate (FF.q, f,f,f,f);
DEFINE
  in1_A   := buf_in1_A1.val;
  out1_R  := buf_out1_R1.val;
  f       := FALSE;
  t       := TRUE;
FAIRNESS running


MODULE environment (in1_A, out1_R)
VAR
  ENV_in1   : process cgate (!in1_A, f,t,f,f);
  ENV_out1 : process cgate (out1_R, f,t,f,f);
DEFINE
  in1_R   := ENV_in1.val;
  out1_A  := ENV_out1.val;
  f       := FALSE;
  t       := TRUE;
FAIRNESS running
```

Figure 6 (top) Click Storage circuit and environment models from Figure 1 (right-column-top) with corresponding module definitions for the Model Checker Library (bottom) coded in NuSMV. In the picture, the gates for *ENV_in1* and *ENV_out1* contain the letter “L” to indicate that they are lazy. The code for *cgate* and *ff_posedge* follows in Figure 7.

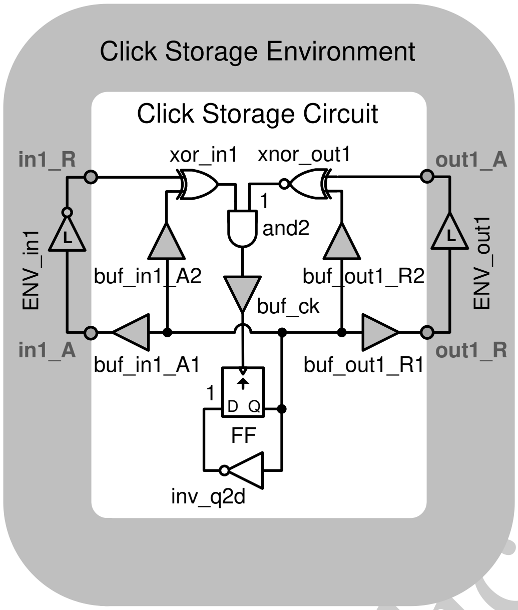
```
MODULE cgate (set, init_val, lazy, stop_rise, stop_fall)
VAR
  val : boolean;
  semimodular : boolean;
ASSIGN
  init(val)         := init_val;
  init(semimodular) := TRUE
  next(val) := case
    (stop_rise & !val & set) | (stop_fall & val & !set) : val;
    lazy                                                : {val, set};
    TRUE                                                : set;
  esac;
TRANS
  next(semimodular) = case
    ((!stop_rise & !val & set) | (!stop_fall & val & !set))
              & next(val)=next(set) & next(val)=val : FALSE;
    TRUE                                            : semimodular;
  esac;
--PROPERTIES for "digital health"
  CTLSPEC AG semimodular

MODULE ff_posedge(ck, d, init_q)
VAR
  q : boolean;
ASSIGN
  init(q) := init_q;
TRANS
  next(q) = case
    !ck & next(ck) : d;
    TRUE           : q;
  esac;
```

Figure 7 Module definitions for *cgate* and *ff_posedge*. Earlier commands in a case statement have a higher priority. NuSMV uses the symbol "!" for logical negation.



```
MODULE main
VAR
  ComponentProtocol      : protocol (in1_R, in1_A, out1_R, out1_A);
  ComponentCircuit       : process circuit (in1_R, out1_A);
  ComponentEnvironment : process environment (in1_A, out1_R);
DEFINE
  in1_R   := ComponentEnvironment.in1_R;
  in1_A   := ComponentCircuit.in1_A;
  out1_R  := ComponentCircuit.out1_R;
  out1_A  := ComponentEnvironment.out1_A;
FAIRNESS running
```

Figure 8 Copy of the Model Checker Netlist in Figure 4 without timing constraints (left) and corresponding NuSMV code (right) with one instance each of the component's protocol, circuit, and environment. The code of each instance can be found in Figures 5–6.



"Digital Health" Failure

Initial State:
state=s0
Run step 1:
ENV_in1.val+
in1_R+
state=s1
Run step 2:
xor_in1.val+
Run step 3:
and2.val+
Run step 4:
buf_ck.val+
FF.q+
Run step 5:
buf_out1_R1.val+
out1_R+
state=s3
Run step 6:
buf_out1_R2.val+
Run step 7:
ENV_out1.val+
out1_A+
state=s4
xnor_out1.semimodular=FALSE

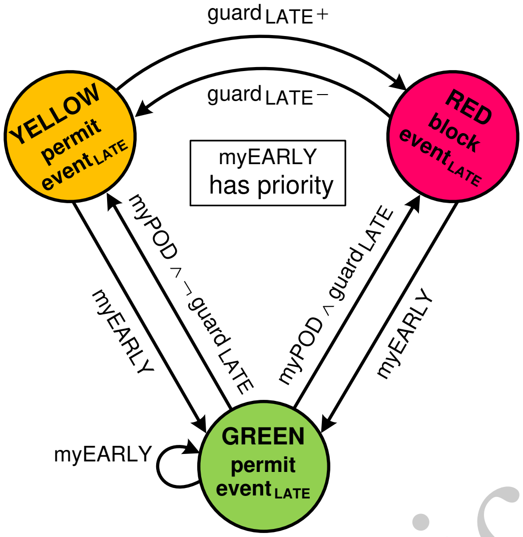
Failure: CTLSPEC AG semimodular

Protocol Failure

Initial state:
state=s0
Run step 1:
ENV_in1.val+
in1_R+
state=s1
Run step 2:
xor_in1.val+
Run step 3:
and2.val+
Run step 4:
buf_ck.val+
FF.q+
Run step 5:
buf_out1_R1.val+
out1_R+
state=s3
Run step 6:
buf_out1_R2.val+
Run step 7:
xnor_out1.val−
Run step 8:
and2.val−
Run step 9:
buf_ck.val−
Run step 10:
inv_q2d−
Run step 11:
ENV_out1.val+
out1_A+
state=s4
Run step 12:
xnor_out1.val+
Run step 13:
and2.val+
Run step 14:
buf_ck.val+
FF.q−
Run step 15:
buf_out1_R1.val−
out1_R−
state=errorOUT

Failure: CTLSPEC AG state!=errorOUT

Figure 9 (left) Copy of the Click Storage circuit and environment coded in Figures 6–7 and two counterexamples (right) showing a "digital health" failure for gate *xnor_out1* and a failure in state *s4* of the protocol (Figure 5) monitoring the circuit and environment. The counterexamples each describe a path of events through the circuit. An event is a rising or falling signal transition. The counterexamples indicate rising signal transitions by appending the symbol "+" to the signal name, like ENV_in1.val+ in step 1. They indicate falling transitions by appending the symbol "−," like out1_R− in step 15 of the path with the protocol failure.



Initial Set of Relative Timing Constraints
for the Click Storage Circuit and Environment in Figure 6

myName		myPOD		myEARLY		myLATE
rt1	:	($\underline{\neg\text{FF.q}}$ ∧ and2.val+)	→	xor_in1.val−	<	in1_R−
rt2	:	($\underline{\text{FF.q}}$ ∧ and2.val+)	→	xor_in1.val−	<	in1_R+
rt3	:	($\underline{\neg\text{FF.q}}$ ∧ and2.val+)	→	xnor_out1.val−	<	out1_A+
rt4	:	($\underline{\text{FF.q}}$ ∧ and2.val+)	→	xnor_out1.val−	<	out1_A−
rt5	:	and2.val+	→	xor_in1.val−	<	xnor_out1.val+
rt6	:	and2.val+	→	xnor_out1.val−	<	xor_in1.val+
rt7	:	and2.val+	→	and2.val−	<	($\underline{\text{xor\_in1.val}}$ ∧ xnor_out1.val+)
rt8	:	and2.val+	→	and2.val−	<	($\underline{\text{xnor\_out1.val}}$ ∧ xor_in1.val+)
rt9	:	and2.val+	→	buf_ck.val−	<	and2.val+
rt10	:	FF.q+	→	inv_q2d.val−	<	buf_ck.val+
rt11	:	FF.q−	→	inv_q2d.val+	<	buf_ck.val+

Figure 10 (left) *Stoplight* model of a relative timing constraint for use by the model checker, and (right) initial set of relative timing constraints for the Click Storage component derived by failure analysis of counterexamples for the NuSMV netlist in Figure 8. Failure analysis of the two counterexamples from Figure 9 in Section III-C1 gave us *rt3* and *rt5* — and implicitly *rt1* to *rt6*. Constraint *myNAME : myPOD* → *myEARLY* < *myLATE* expresses that after *myPOD* the computation encounters *myEARLY* before *myLATE*. The expressions *myPOD*, *myEARLY*, and *myLATE* formulate guarded events, as explained in the text of Section III-C2. The expressions for the guards are underlined. Rising events end with the symbol "+" and falling events end with the symbol "−."

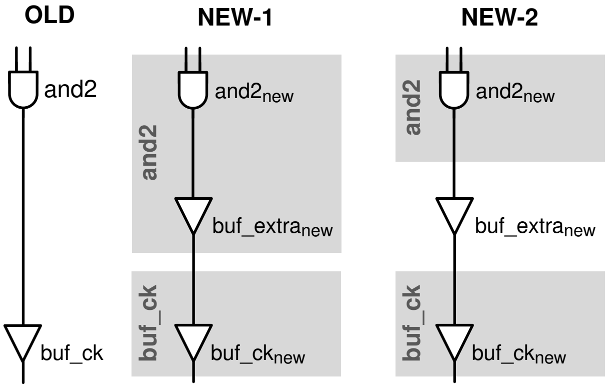


Figure 11 (left) Click Storage sub-circuit from Figure 6, and (right) two new post-layout versions with an extra buffer. Sub-circuit NEW-1 groups *and2*$_{new}$ and *buf_extra*$_{new}$ and identifies the group with *and2* in OLD. The semimodular gate behavior of each gate is covered if the semimodularity of *and2* is covered. Constraints *rt7* and *rt8* of Figure 10 cover the semimodularity of *and2*, and thus that of *and2*$_{new}$ and *buf_extra*$_{new}$ in NEW-1. The grouping and renamings for sub-circuit NEW-2, however, keep *buf_extra*$_{new}$ isolated. Because *rt1* to *rt11* are the weakest possible constraints for the original circuit with sub-circuit OLD, they fail to cover the semimodular gate behavior of *buf_extra*$_{new}$ in NEW-2. Because the timing patterns *p5* and *p6* in Figure 12 cover the semimodular behavior of all gates from *and2* through *buf_ck*, they cover the semimodular behavior of the intermediary gate, *buf_extra*$_{new}$, in both NEW-1 and NEW-2.

Timing Patterns
replacing rt1 to rt11 from Figure 10

myName		myPOD		myEARLY		myLATE
p1	:	and2.val+	→	xor_in1.val−	<	in1_R±
p2	:	and2.val+	→	xnor_out1.val−	<	out1_A±
p3	:	and2.val+	→	xor_in1.val−	<	out1_A±
p4	:	and2.val+	→	xnor_out1.val−	<	in1_R±
p5	:	and2.val+	→	buf_ck.val−	<	in1_R±
p6	:	and2.val+	→	buf_ck.val−	<	out1_A±
p7	:	and2.val+	→	inv_q2d.val±	<	in1_R±
p8	:	and2.val+	→	inv_q2d.val±	<	out1_A±

Parametrized Timing Patterns
for N incoming and M outgoing channels

$(0 < \mathbf{n1}, \mathbf{n2} \leq N \quad \text{and} \quad 0 < \mathbf{m1}, \mathbf{m2} \leq M)$

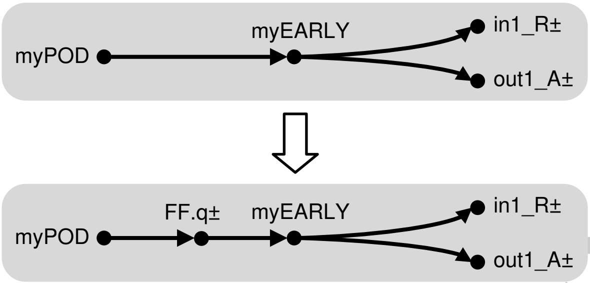
myPOD	myEARLY	myLATE

$$\text{p} \;:\; \text{and2.val}+ \;\rightarrow\; \left\{\begin{array}{c}\text{xor\_in}\,\mathbf{[n1]}\,\text{.val}-\\ \text{xnor\_out}\,\mathbf{[m1]}\,\text{.val}-\\ \text{buf\_ck.val}-\\ \text{inv\_q2d.val}\pm\end{array}\right\} < \left\{\begin{array}{c}\text{in}\,\mathbf{[n2]}\,\text{\_R}\pm\\ \text{out}\,\mathbf{[m2]}\,\text{\_A}\pm\end{array}\right\}$$

Figure 12 Timing Patterns for a Click Storage component with a single incoming and a single outgoing channel (top). Pattern *p* (bottom) re-phrases and parametrizes *p1* to *p8* to multiple incoming and outgoing channels. It expresses that after *myPOD* the computation must satisfy all *myEARLY* before any *myLATE*. Symbols "+," "−" and "±" at the end of a signal indicate a rising, falling, or either signal transition.

```
MODULE rt (eventPOD, eventEARLY, init_rt, guardPOD, guardEARLY, guardLATE, xPOD, xEARLY)
VAR
  stoplight : {GREEN, YELLOW, RED};
ASSIGN
  init(stoplight) := init_rt;
TRANS
  next(stoplight) = case
    myEARLY                                : GREEN;
    stoplight=GREEN & myPOD & !guardLATE: YELLOW;
    stoplight=GREEN & myPOD & guardLATE : RED;
    stoplight=YELLOW & next(guardLATE)     : RED;
    stoplight=RED & next(!guardLATE)       : YELLOW;
    TRUE                                   : stoplight;
  esac;
DEFINE
  myPOD     := guardPOD & ((xPOD & eventPOD!=next(eventPOD)) | (!eventPOD & next(eventPOD)))
  myEARLY := guardEARLY & ((xEARLY & eventEARLY!=next(eventEARLY)) | (!eventEARLY & next(eventEARLY)))
  stop         := (stoplight=RED);


MODULE circuit (in1_R, out1_A)
-- RELATIVE TIMING CONSTRAINTS
VAR
  p1p3: rt (and.val, !xor_in.val, GREEN, t,t,t,f,f);
  p2p4: rt (and.val, !xnor_out.val, GREEN, t,t,t,f,f);
  p5p6: rt (and.val, !ckbuf.val, GREEN, t,t,t,f,f);
  p7p8: rt (and.val, inv_q2d.val, GREEN, t,t,t,f,t);
DEFINE
  stop_in1_R_x   := p1p3.stop | p2p4.stop | p5p6.stop | p7p8.stop;
  stop_out1_A_x := p1p3.stop | p2p4.stop | p5p6.stop | p7p8.stop;
--Insert lines 2 to 18 from MODULE circuit of Figure 6


MODULE environment (in1_A, out1_R, stop_in1_R_x, stop_out1_A_x)
VAR
  ENV_in1  : process cgate (!in1_A, f,t, stop_in1_R_x, stop_in1_R_x);
  ENV_out1 : process cgate (out1_R, f,t, stop_out1_A_x, stop_out1_A_x);
--Insert lines 25 to 30 from MODULE environment of Figure 6


MODULE main
VAR
  ComponentProtocol       : protocol (in1_R, in1_A, out1_R, out1_A);
  ComponentCircuit        : process circuit (in1_R, out1_A);
  ComponentEnvironment : process environment (in1_A, out1_R, stop_in1_R_x, stop_out1_A_x);
DEFINE
  stop_in1_R_x   := ComponentCircuit.stop_in1_R_x;
  stop_out1_A_x := ComponentCircuit.stop_out1_A_x;
--Insert lines 7 to 11 from MODULE main of Figure 8
```

Figure 13 NuSMV model checker code changes for adding the relative timing constraints captured in patterns *p1* to *p8* of Figure 12. The Model Checker Library adds the module definitions for *protocol* (Figure 5) and logic gates *cgate* and *ff_posedge* (Figure 7). **Note:** In NuSMV, earlier commands in a case statement have a higher priority, and the symbol "!" is used for logical negation.



Parametrized Timing Patterns for STA code generation replacing p from Figure 12 ($\textbf{guard} \overset{def}{=}$ sta1.stoplight $\neq$ GREEN)

	myPOD		**myEARLY**		**myLATE**
sta1 :	and2.val+	→	FF.q±	<	$\left\{ \begin{array}{c} \text{in}\,\textbf{[n2]}\,\text{\_R}\pm \\ \text{out}\,\textbf{[m2]}\,\text{\_A}\pm \end{array} \right\}$
sta2 :	$(\textbf{guard} \wedge \text{FF.q}\pm)$	→	$\left\{ \begin{array}{c} \text{xor\_in}\,\textbf{[n1]}\,\text{.val}\pm \\ \text{xnor\_out}\,\textbf{[m1]}\,\text{.val}\pm \\ \text{buf\_ck.val}\pm \\ \text{inv\_q2d.val}\pm \end{array} \right\}$	<	$\left\{ \begin{array}{c} \text{in}\,\textbf{[n2]}\,\text{\_R}\pm \\ \text{out}\,\textbf{[m2]}\,\text{\_A}\pm \end{array} \right\}$

Figure 14 We modify timing pattern *p* of Figure 12 to simplify its STA code. Knowing that all *p*'s *myEARLY* paths go through flipflop *FF* compensates the need to differentiate rising from falling transitions in *myEARLY*. The lefthand graphs show the event orderings specified by *p* before (top) and after (bottom) adding *FF*. The righthand constraints *sta1* and *sta2* formulate the modified version of *p*, and have been verified by using their NuSMV translations instead of those of *p* in lines 24–30 of Figure 13. The guard in *myPOD* of *sta2* plays the role of the baton in a relay race, handing over the task of blocking *myLATE* from *sta1* to *sta2*.