Asynchronous Circuit Verification Using Trace Theory and CCS

Ganesh Gopalakrishnan *
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112
ganesh@bliss.utah.edu

June 29, 1995

Abstract

We investigate asynchronous circuit verification using Dill's trace theory *[1] as well as Milner's CCS (as mechanized by the Concurrency Workbench). Trace theory is a formalism specifically designed for asynchronous circuit specification and verification. CCS is a general purpose calculus of communicating systems that is being recently applied for hardware specification and verification [2]. Although both formalisms are similar in many respects, we find that there are many interesting differences between them when applied to asynchronous circuit specification and verification. The purpose of this paper is to point out these differences, many of which are precautions for avoiding writing incorrect specifications. A long-term objective of this work is to find a way to take advantage of the strengths of both the Trace Theory verifier and the Concurrency Workbench in verifying asynchronous circuits.*

1 Introduction

As VLSI systems become larger, faster, and more complex, timing problems in them become progressively more severe, and account for an ever increasing percentage of their design and debugging expenses. One emerging solution

*Supported in part by NSF Award MIP-8902558.

to these problems lies in adopting an asynchronous style of design. Asynchronous circuits have a number of strengths, the principle ones being that of modularity and incremental expandability.

Although asynchronous circuit design techniques have been known for nearly four decades, and their advantages have been widely discussed, they have not been adopted widely for several reasons. The most important reason is the inadequacy of design formalisms as well as tools to deal with the concurrency exhibited by asynchronous circuits. The situation has recently been changing, with the development of asynchronous circuit compilers [3, 4, 5, 6, 7] as well as formalisms, the principal ones being several *trace theories*, notably those of Dill [1] and Ebergen [8]. In addition, popularizing lectures such as Sutherland's 1988 Turing award lecture [9] have helped. See [10] for a survey.

I have been studying Dill's trace theory for some time now (referred to in the rest of this paper simply as "trace theory"). I also am fairly familiar with Milner's Calculus of Communicating Systems. For a while I believed that trace theory, being a formalism tailored specifically for studying asynchronous circuits, is a "safer bet" in terms of the *direct correspondence* that its constructs have to actual circuit phenomena such as transistors going on/off, gates firing, etc. This correspondence is very important because humans are no longer able to reason directly in terms of low-level circuit phenomena because of the increasing circuit complexity. If there is even the slightest risk of mismatch between the abstractions offered by the formalism and the circuit realities, one's reasoning can go way off course before one so realizes.

The asynchronous circuits considered in this paper are assumed to follow the *transition signaling* discipline [9]: a module toggles the current logic level of a wire "a" in order to invoke input action "a" of the recipient.)

My main reason for thinking that CCS is not a suitable formalism for studying asynchronous circuits, in the light of what I just now said, was based on the fundamental difference in the way communication is modeled in CCS versus how it is modeled in trace theory. Asynchronous circuits communicate over wires. Communication over wires causes information to flow only in one direction: the receiver knows when it receives the communication; however, the sender does not know when the receiver receives the communication. In CCS, information flow during communication is *bidirectional* because of the "handshake" or "rendezvous" semantics (both the sender and the receiver know that the other has received the communication before they proceed). Said another way, in asynchronous circuits, a module *cannot refuse an input* simply because the sender does not sense the *receptiveness* of the receiver before it sends a communication. In CCS, since receptiveness is explicitly checked for during handshake, the inputs offered by the sender can be refused by a potential receiver.

My opinions in this regard have recently changed as I have been noticing several researchers use either CCS or CCS-like formalisms for modeling asynchronous circuits. Two examples are the use of CCS by Aldwinckle, Nagarajan, and Birtwistle [2], and the use of CIRCAL by Bailey and Milne [11]. This trend is quite important because this way one could "re-use" what is being developed in the world of CCS (for example, tools such as the Concurrency Workbench, "CWB") for verifying asynchronous circuits.

In this paper, I report results from my preliminary studies in applying both Dill's *trace theory* [1] as well as Milner's CCS [12] (as mechanized by the CWB) to verify asynchronous circuits. Although both formalisms are similar in many respects, I find that there are many interesting differences between them when applied to asynchronous circuit specification and verification. The purpose of this paper is to point out these differences, *many of which are precautions for avoiding writing incorrect specifications.* A long-term objective of this work is to find a way to take advantage of the strengths of both the Trace Theory verifier and the CWB in order to verify asynchronous circuits.

Section 2 is devoted to explaining trace theory, as it may not be well known outside the area of asynchronous design. Familiarity with CCS is assumed. Section 3 explains the problems one may face, if the fact that asynchronous circuits cannot refuse their inputs is ignored. Section 4 explains the problems one may face if a phenomenon called *autofailures* is ignored. Section 5 presents examples where the strengths of the CWB are pointed out. In particular, we establish correctness properties of a new component that we have developed – a *lockable C element.* Section 6 has our conclusions.

2 Background: Trace Theory

2.1 Definitions and Trace Structures

The following definitions and notations are taken from [1]. *Trace theory* is a formalism for modeling, specifying, and verifying speed-independent circuits. It is based on the idea that the behavior of a circuit can be described by a regular set of *traces*, or sequences of transitions. Each trace corresponds to a partial history of signals that might be observed at the input and output terminals of a circuit.

A *simple prefix-closed trace structure*, written *SPCTS*, is a three tuple (I, O, S) where I is the *input alphabet* (the set of input terminal names), O is the output alphabet (the set of output terminal names), and S is a prefix-closed regular set of strings over $\alpha = I \cup O$ called the *success* set. I and O are disjoint. In the following discussion, we assume that S is a non-empty set.

These trace structures are more aptly called *directed* trace structures because the direction (input or output) of every member of a trace is important (as will become clear as we go along). Basically, information flow among circuit modules is *unidirectional* as pointed out before, whereas in CCS (or in other rendezvous based languages) the information flow is *bidirectional*. This distinction has, in fact, been studied extensively by Chen, Udding and Verhoeff in [13] who call it the *synchronous game* and the *asynchronous game*. We show later that ignoring this difference may have dire consequences in terms of not being able to spot certain errors.

We associate a SPCTS with a module that we wish to describe. Roughly speaking, the success set of a module described through a SPCTS is the set of traces that can be observed when the circuit is "properly used".

With each module, we also associate a *failure* set, F, which is a regular set of strings over α. The failure set of a module is the set of traces that correspond to "improper uses" of the module. The failure set of a module is completely determined by the success set: $F = (SI - S)\alpha^*$. Intuitively, $(SI - S)$ describes all strings of the form xa, where x is a success and a is an "illegal" input signal (see below). Such strings are the minimal possible failures, called *chokes*. Once a choke occurs, failure cannot be prevented by future events; therefore F is suffix-closed.

As an example, consider the SPCTS associated with a unidirectional WIRE with input a, output b, and success set

$$(\{a\}, \{b\}, \{\epsilon, a, ab, aba, \ldots\}).$$

The success set is a record of all the partial histories (including the empty one, ϵ), of successful executions of WIRE. An example of a choke for WIRE is the trace "aa". Once input "a" has arrived, a second change in "a" is illegal since it may cause unpredictable output behavior.

There are two fundamental operations on trace structures: *compose* ($\|$) finds the concurrent behavior of two circuits that have some of their terminals of opposite directions (the directions are input and output) connected, and *hide* makes some terminals unobservable (suppressing irrelevant details of the circuit's operation). A third operation, *rename*, allows the user to generate modules from templates by renaming terminals. Details about these operations are reported in [1]; briefly, *compose* is like conjunction; it constructs the success set of the composite as follows. It first takes the Kleene star of the union of the alphabets of the trace structures and then retains from it only strings s such that the projection of s onto the alphabet of trace structure i of the composition (denoted by T_i) is a member of the success set of T_i. After determining the success set of the composite this way, the success set and the failure set are "adjusted" through *autofailure manifestation* and *failure exclusion* as explained in section 2.2. *Compose* of two trace structures $T_1 = (I_1, O_1, S_1)$ and $T_2 = (I_2, O_2, S_2)$ is illegal if $O_1 \cap O_2 \neq \emptyset$; if not, the output alphabet of the composite is $O_1 \cup O_2$ and the input alphabet is $(I_2 \setminus O_1) \cup (I_1 \setminus O_2)$.

Hiding is allowed *only* on the output symbols. If t is a member of $S \cup F$ of trace structure T, then t' is a member of $hide(H)(T)$ where t' is a projection of t onto the alphabet of T. *Hiding* is *not* allowed on input symbols mainly because a module *cannot refuse an input from being applied to it* (and therefore it is hard to define what hiding an input means). However, inputs are effectively "removed" through *compose* because when an input port is connected to an output port, the result is an *output* port.

Rename renames the ports used in a description, mainly to model electrical connections; two ports that are named alike are connected, provided that they are not both outputs.

We can denote the success set of a SPCTS by using state-transition specifications. The success set of WIRE, described earlier, is captured by the following specification, where WIRE is regarded as a *process*:

$$WIRE \quad = \quad a? \rightarrow b! \rightarrow WIRE$$

In a process description, we use '|' to denote *choice*, '$\rightarrow$' to denote *sequencing*, and a *system of tail recursive equations* to capture repetitive behavior. We use symbols such as $a?$ to denote incoming transitions (rising or falling) and $b!$ to denote outgoing transitions (rising or falling). (Extensions to this syntax will be introduced as required.)

When we specify a SPCTS, we generally specify only its success set; its input and output alphabet are usually clear from the context, and hence are left out.

2.2 "Illegal Inputs"

Suppose for a trace structure (I, O, S) with failure set F, $x \in S$ but $xo \in F$ where $o \in O$. Intuitively, after having seen x, the module has an output o which it can autonomously perform, leading to a failure. It is also possible that after x another output $o^{'}$ is enabled which can evade this failure (*i.e.* $xo^{'}$ is a success). Likewise, an input i can also be enabled which, if "applied soon enough" can also evade failure (*i.e.* xi is a success). Nevertheless, having seen x, there is a definite possibility that the module can perform o and fail. Keeping this in mind, we remove x from S and add it to F. (Remember that F has to be made suffix-closed.) x is called an *autofailure*. The process of removing x from S and adding it to F is called *autofailure manifestation*. After autofailure manifestation, S is set to $S \setminus F$; this step is called *failure exclusion*.

The trace structures considered in the rest of this paper are already assumed to have been subject to autofailure manifestation.

2.3 Conformance: The Ability to Perform Safe Substitutions

A trace structure specification, T_S, can be compared with a trace structure description, T_I, of the actual behavior of a circuit. When T_I implements T_S, we say that T_I *conforms to* T_S; that is, $T_I \preceq T_S$. (The inputs and outputs of the two trace structures must be the same.) This relation is a preorder and is called *conformance*. *Conformance* holds when T_I can be *safely substituted* for T_S.

More precisely, $T_I \preceq T_S$ if for every T', whenever $T_S \parallel T'$ has no failures, $T_I \parallel T'$ has no failures, either. Intuitively, T_I:

(a) must be able to handle every input that T_S can handle (otherwise, T_I could fail in a context where T_S would have succeeded); and

(b) must not produce an output unless T_S produces it (otherwise, T_I could cause a failure in the surrounding circuitry when T_S would not).

We illustrate these two facets of *conformance*, first considering restrictions on input behavior (case (a)). Consider a JOIN element:

$$\begin{aligned} J \quad = \quad & a? \rightarrow b? \rightarrow c! \rightarrow J \\ & \mid b? \rightarrow a? \rightarrow c! \rightarrow J \end{aligned}$$

Now, consider a modified JOIN:

$$J1 \quad = \quad a? \rightarrow b? \rightarrow c! \rightarrow J1$$

Notice that the *success set* of $J1$ leaves out the trace $b; a; c$. Clearly it is not safe to substitute $J1$ for J: $J1$ cannot accept a transition on b as its first input, whereas the environment is allowed to generate a b as its first output transition, because this would have been acceptable for J. Formally, we say $J1 \not\preceq J$, since the implementation cannot accept an input transition which the specification can receive.

However, note that it *is* safe to substitute J for $J1$, since J can handle every input (and more) which $J1$ can handle; so $J \preceq J1$. *Trace theory allows an implementation to have "more general" input behavior than its specification.*

Next, consider the case of restrictions on output behavior (case (b) above). We begin with a simple case:

$$CONCUR\_MOD = \; a? \to (b'! \parallel c'!) \to CONCUR\_MOD$$
$$SEQNTL\_MOD = \; a? \to b'! \to c'! \to SEQNTL\_MOD$$

Note that the *success set* of $SEQNTL\_MOD$ omits the trace $a; c$. It is not safe to substitute $CONCUR\_MOD$ for $SEQNTL\_MOD$: some environment of $SEQNTL\_MOD$ may not accept a transition on c after producing an a. Therefore, $CONCUR\_MOD \not\preceq SEQNTL\_MOD$ (intuitively, implementation $CONCUR\_MOD$ is "too concurrent").

However, $SEQNTL\_MOD$ *can* be safely substituted for $CONCUR\_MOD$ in *any* environment. Any environment accepting outputs from $CONCUR\_MOD$ will also accept outputs generated by $SEQNTL\_MOD$, so $SEQNTL\_MOD \preceq CONCUR\_MOD$. *Trace theory allows an implementation to have "more constrained" output behavior than its specification.*

This point can be illustrated more dramatically. We return to the earlier JOIN and a new implementation:

$$\begin{aligned} AlmostWood \;\; = & \;\;\; a? \to b? \to c! \to AlmostWood \\ & \;\; \mid b? \to a? \to AlmostWood \end{aligned}$$

The reason why J can be safely substituted by $AlmostWood$ in any context is the following. So long as the environment and the component keep generating the sequence $abcabcabc\ldots$, both J and $AlmostWood$ behave alike. Suppose the environment generates the string ba and awaits a c. J does generate a c after seeing ba, thereby allowing the environment to proceed; $AlmostWood$, on the other hand, outputs nothing, and awaits a further a or a b—at the same time as the environment is awaiting a c; in this case, the result is a deadlock.

Going to the extreme, we find that

$$\begin{aligned} \mathit{BlockOfWood} &= \quad a? \to \mathit{BlockOfWood} \\ &\quad \mid b? \to \mathit{BlockOfWood} \end{aligned}$$

conforms to J.

In summary, *conformance* allows an implementation to be a *refinement* of a specification: an implementation may have "more general" input behavior or "more constrained" output behavior than its specification. However, we want to show not only that an implementation does no harm, but that it also does something useful! Unfortunately, prefix-closed trace theory cannot distinguish "constrained" output behavior from deadlock. In spite of the usefulness of trace theory, this is its greatest practical weakness.

2.4 On Establishing Conformance

A verifier has been developed by Dill to establish conformation. Relation $\preceq$ is established in this verifier as follows (we use T, T_S, *etc.* to denote trace structures):

- The verifier constructs a trace structure, $\overline{T_S}$, called the *mirror* of specification T_S (see [1]; originally proposed in [8]). $\overline{T_S}$ is the same as T_S, but with input and output sets reversed. The mirror is the worst-case environment which will "break" any trace structure that is not a true implementation of T_S.
- The verifier then generates the parallel composition of the implementation, T_I, and the mirror, $\overline{T_S}$: $T_I \parallel \overline{T_S}$. It has been proven that $T_I \preceq T_S$ iff $T_I \parallel \overline{T_S}$ is failure-free (see [1]).
- $T_I \preceq T_S$ is checked by testing that $T_I \parallel \overline{T_S}$ is free of failures. This check can be performed by "simulating" the parallel behavior of the two trace structures, presented in Figure 1.

As an example of the above simulation, consider the simulation of $J1$ against $\overline{J}$, where $\overline{J}$ is the mirror of specification J:

$$\begin{aligned} \overline{J} &= \quad a! \to b! \to c? \to \overline{J} \\ &\quad \mid b! \to a! \to c? \to \overline{J} \end{aligned}$$

We can see that $\overline{J}$ is the only module capable of performing the first output action: either $a!$ or $b!$. The production of $b!$ will cause $J1$ to choke.

2.5 Conformation Equivalence

We have seen that while *conformance* captures the notion of "refinement", it cannot capture the notions of deadlock and livelock. There is another relation that can be considered: *conformation equivalence.* Trace structures A and B are *conformation equivalent* ($A \overset{conf}{\equiv} B$) if $A \preceq B$ and $B \preceq A$ (see [1]).

Unfortunately, just as *conformance* is "too weak" a relation for our purposes, *conformation equivalence* is often "too strong". Often, for a specification $Spec$ and implementation Imp, where $Imp \preceq Spec$, we cannot establish that $Imp \overset{conf}{\equiv} Spec$. For example, Imp commonly is *overbuilt* in the sense that it accepts more inputs than necessary.

Such an implementation gives rise to the following problems. In showing $Imp \preceq Spec$, no problem arises, because Imp will accept all the inputs that $Spec$ can. However, in trying to show that $Spec \preceq Imp$, we "simulate" $\overline{Imp} \parallel Spec$. Since Imp can accept more inputs than it needs to, $\overline{Imp}$ ends up generating more outputs than it "needs to"—some of these outputs go beyond what $Spec$ can accept, and thus the test $Spec \preceq Imp$ fails.

How do we rescue the situation? The answer lies in *not* attempting $Spec \preceq Imp$, but merely whether $S_{Spec} \subseteq S_{Imp}$, where '$S_M$' denotes the success set of 'M'. We have identified precisely such a relation, called *strong conformance* [14]. This relation is now briefly explained.

2.6 Strong Conformance

Definition: We define $T \sqsubseteq T'$, read T conforms strongly to T', if $T \preceq T'$ and $S_T \supseteq S_{T'}$. The algorithm to check for strong conformance is omitted to conserve space.

The *strong conformance* relation is *safe* in that it guarantees conformance. However, it is *not* guaranteed to catch all liveness failures; but for a number of examples, a verifier based on strong conformance provides much better error detection capabilities [14].

3 Examples Motivating Non-refusal of Inputs

Having studied Dill's trace theory, we now proceed to experiment with the CWB, and compare our observations with those observed in Dill's trace theory verifier.

Consider the process *WIRE* defined on page 5. Suppose we specify this process in CCS as

```
Wire = a.'b.Wire
```

Here, following the syntax of the Concurrency Workbench [15], an action of the form `'x` is a *co-name* (output action) and an action of the form `x` is an input action. Let us pose the question: "does `Wire` conform to *Spec*, where:

```
Spec = a.('b.Spec + a.Spec)
```

In other words, we are asking whether `Wire` is a safe substitution (in the sense of conformance), for `Spec`, in any context. The most liberal environment in which `Spec` can be operated is obtained by taking its mirror:

```
Specmirror1 = 'a.(b.Specmirror1 + 'a.Specmirror1)
```

Though the above process is the mirror, for practical reasons, we modify it to `Specmirror` given below. Since CCS converts synchronizing actions to a silent action, *tau* (written `t` in our syntax), we add "marker" actions `aout` and `bout` to `Specmirror1` so that its execution can be more meaningfully observed from outside. We thus obtain:

```
Specmirror = 'a.aout.(b.'bout.Specmirror + 'a.aout.Specmirror)
```

Now consider the system:

```
Specmirror_Wire = (( Specmirror | Wire)\ {a,b})[a/aout,b/bout]
```

In the combined system, after accepting an '`a`', `Wire` can be subject to another '`a`' from `Specmirror`; however, `Wire` can refuse this '`a`' and proceed to do a '`b`'. Therefore, the combined system does *not* exhibit a deadlock (as revealed by the "find deadlock" – `fd` – command).

```
fd Specmirror_Wire

No such agents.
```

If the above specifications are transliterated into trace-theoretic specifications and we ask if `Wire` conforms to `Spec`, the answer will be *false*, meaning that a *choke* '`a,a`' can occur!

In other words, when modeling asynchronous circuits, it can be dangerous (in the sense of not being able to detect certain chokes) *not* to take into account the fact that asynchronous circuits can ignore their inputs.

How do we model a "real wire"? The fact that an actual wire cannot refuse an input is easily captured by amending the specification of `Wire` to `Realwire`, below. Then we define `Specmirror_Realwire`, also defined below, which indeed reveals *precisely* the choke discovered by the trace theory verifier:

```
Realwire            = a.('b.Realwire + a.Choke)
Choke               = nil
Specmirror_Realwire = (( Specmirror | Realwire)\ {a,b})[a/aout,b/bout]

fd Specmirror_Realwire

--- t a t a ---> (Specmirror | Choke)\{a,b}[a/aout,b/bout]
```

This shows that `Specmirror_Realwire` "deadlocks" by going into state `Choke`. (The definition of state `Choke` helps us spot these deadlocks more easily. Also the above "deadlock" is merely a way to model actual chokes in circuits; we could very well have modeled a choke through any other error situation that is easily detectable in the CWB.)

Thus, it appears that `Realwire` models an electrical conductor more faithfully. We shall confirm this through another experiment presented later.

4 Dealing With Autofailures

To motivate the importance of *directionality* in trace theory, as well as the phenomenon of *autofailures*, consider the following CCS definitions:

```
Test1   =  c.'d.'e.Test1
Driver1 = 'c.d.Driver1

Test2   = 'c.d.'e.Test2
Driver2 =  c.'d.Driver2

System1 = (Test1 | Driver1)\{c,d}
System2 = (Test2 | Driver2)\{c,d}
```

The only difference between `System1` and `System2` is that their constituent processes use different directions for their ports `c` and `d`. Doing so has *no observable effect* on the computations of `System1` and `System2` because the ports `c`, `c'`, `d`, and `d'` synchronize in the same fashion as before, and they are unobservable. In other words, we could show that `System1` and `System2` are observationally congruent.

Now, suppose we *transliterate* these specifications into the input syntax of Dill's verifier. In other words,

$$\begin{aligned} Test1 &= c? \rightarrow d! \rightarrow e! \rightarrow Test1 \\ Driver1 &= c! \rightarrow d? \rightarrow Driver1 \\ \\ Test2 &= c! \rightarrow d? \rightarrow e! \rightarrow Test2 \\ Driver2 &= c? \rightarrow d! \rightarrow Driver2 \\ \\ System1 &= hide(c, d)(compose(Test1, Driver1)) \\ System2 &= hide(c, d)(compose(Test2, Driver2)) \end{aligned}$$

We find that *System*1 exhibits a *choke* while *System*2 doesn't!

Here is the reason. Consider *System*1. First *Driver*1 applies a *c!* onto *Test*1 causing both the modules to make progress; then *Test*1 applies a *d!* onto *Driver*1, causing *Driver*1 to return to its top state, while *Test*1 is in a state where it can only generate output *e!*. If *Test*1 were to now generate *e!* "soon enough", it would return to its top-level state, and all would be well (both processes resume their behavior); however if *Test*1 were a bit slow relative to *Driver*1, the latter, since it is now in its top level state, would apply a *c!* which *Test*1 cannot accept! The simulation of *System*2 is safe because after *Driver*2 is back in its top level state, it only *awaits* an input *c?* – and this input *can only come from* Test*2 because the output port* c! *of* Test*2 is connected to input port* c? *of* Driver*2, and there can be no further "drives" onto this node* (see the restrictions on *compose*).

How do we make the CCS specifications manifest these errors? There are two approaches. The first approach consists of the following steps. (1) connect a `Realwire` to the input ports of every component; (2) then assemble the system. For example, define

```
Driver1' = ( Driver1  [rwb_d/d] | Realwire [d/rwa, rwb_d/rwb] ) \ rwb_d
```

and likewise define `Driver2'`, `Test1'`, and `Test2'`. Now we get:

```
eq System1' System2'
false

fd System1'

* --- rwb_c t e rwb_c t t t e rwb_c t t t e rwb_c t e rwb_c --->
((('d.e.Test1)[rwb_c/c] | Choke[rwb_c/rwb,c/rwa]) | (Driver1[rwb_d/d] |
Choke[d/rwa,rwb_d/rwb])\rwb_d)\{c,d}
* --- rwb_c t e rwb_c t t t e rwb_c t e rwb_c t ---> ((('d.e.Test1)[rwb_c/c] |
Choke[rwb_c/rwb,c/rwa]) | ((d.Driver1)[rwb_d/d] | Choke[d/rwa,rwb_d/rwb])\rwb_d)\{c,d}
* --- rwb_c t e rwb_c t e t t ---> ((('d.e.Test1)[rwb_c/c] |
Realwire[rwb_c/rwb,c/rwa]) | ((d.Driver1)[rwb_d/d] |
Choke[d/rwa,rwb_d/rwb])\rwb_d)\{c,d}

fd System2'
No such agents.
```

Notice that we can now detect a choke in `System1'` but not in `System2'`

The second approach (which is automatable and more efficient in practice) is to redefine the processes as follows, which then gives the indicated simulation results:

```
Test1''   =  c.(c.Choke + 'd.(c.Choke + 'e.Test1''))
Driver1'' =  d.Choke + 'c.d.Driver1''

Test2''   =  d.Choke + 'c.d.(d.Choke + 'e.Test2'')
Driver2'' =  c.(c.Choke + 'd.Driver2'')

System1'' = (Test1'' | Driver1'')\{c,d}
System2'' = (Test2'' | Driver2'')\{c,d}

eq System1'' System2''
false

fd System1''
--- t t t ---> (Choke | d.Driver1)\{c,d}

fd System2''

No such agents.
```

The way in which we have modified `Test1`, *etc.*, to `Test1''`, *etc.* is as follows: for every agent, for every reachable state of the agent, if that state has outgoing transitions on inputs $I_m \subseteq I$ where I are all the inputs of the agent, add transitions on inputs $I \setminus I_m$ to state $Choke$. This transformation helps reveal autofailures through the "find deadlock" (`fd`) command. We call this step *adding failure paths.*

To sum up, in this section, we have shown the following:

1. We have shown how autofailures can be detected in the context of the CWB.

2. We have shown how *conformance* can be checked for by explicitly creating the *mirror* of the given specification (for example, see `Specmirror1`). Actually, in effect, *strong conformance* is checked for if the CWB command `eq` is used.

To "simulate" the effects of Dill's trace theory in CCS (and to then use the CWB to detect errors), a few additional transformations are required on CCS agent definitions. Since CCS agent connections are "point-to-point" (*i.e.* a matching name and a co-name are turned into a τ) whereas Dill's trace theory takes the point of view of having "infinite fanout" (*i.e.* an output connected to an input is retained as an output), explicitly use `ForkN` modules in order to fanout the transition of an electrical signal. For example, for a fanout of two, we use the `Fork2` module

```
Fork2 = a.('b1.'b2.Fork2 + 'b2.'b1.Fork2)
```

5 Assorted Examples

In this section we first discuss the verification of a Lockable C Element. Then we discuss some examples pertaining to the detection of deadlocks.

5.1 A Lockable C element

Muller's C element is a very widely used component in asynchronous circuits. It is very close to the *join* element, J, introduced on page 6. Its specification can be expressed in CCS (before the step of adding *failure paths* to avoid clutter) as:

```
* A C element that allows ``double clutching'': e.g. two successive a's cancel.
*
C      = a.Aseen + b.Bseen
Aseen  = a.C + b.ABseen
Bseen  = b.C + a.ABseen
ABseen = 'c.C

* A C element that has seen a ``b''
Cb     = Bseen
```

In typical applications, a `C` element allows two threads of control to rendezvous. One common use of a `C` element is to build a *micropipeline* stage as shown in figure 2.

We have recently developed a *lockable* version of the C element called `LockC`. Its specification is now given.

```
LockC  = a.Aseen + b.Bseen + lock.'lack.Locked
Aseen  = a.LockC + b.ABseen + lock.'lack.AseenLocked
Bseen  = b.LockC + a.ABseen + lock.'lack.BseenLocked
Locked = lock.'lack.LockC + a.AseenLocked + b.BseenLocked
ABseen = a.Bseen + b.Aseen + lock.('lack.ABseenLocked + 'c.'lack.Locked)
                 + 'c.LockC
ABseenLocked = lock.'lack.ABseen + a.BseenLocked + b.AseenLocked
AseenLocked  = a.Locked + lock.'lack.Aseen + b.ABseenLocked
BseenLocked  = b.Locked + lock.'lack.Bseen + a.ABseenLocked

* LockC that has seen a "b"

LockCb = Bseen
```

The basic application of `LockCb` is in building *stallable micropipelines* as described in [16]. It differs from `Cb` in that it offers the possibility of being "locked" via a `lock` signal, and acknowledges locking via `lack`; it is then unlocked via `lock` and it acknowledges the unlocking also via `lack`.

Suppose `LockCb` is used in place of `Cb`, with the `lock` and `lack` of `LockCb` connected to a driver process, as shown in figure 3:

```
LockCDriver = 'lock.lack.'lock.lack.LockCDriver
```

Further assume that `lock` and `lack` are restricted. Then we expect the circuit using `LockCb` to behave exactly the same as the one using `Cb`. We could confirm this using the CWB, using the `eq` command.

We could also apply the `diveq` command which checks whether both the processes are observationally equivalent, also respecting divergence – this proved to be false, because the circuit using `LockCb` can diverge through a sequence of `lock, 'lack` actions – `LockCDriver` can be so fast that it causes the circuit using `LockCb` to diverge in a tau loop, effectively preventing `LockCb` from making any progress.

Finally, we could check the following propositions about the circuit using the CWB: after every `lock`, `'lack` will eventually happen; and vice versa. These model checking commands are quite valuable in verifying asynchronous circuits. Currently this facility is not available in Dill's trace theory verifier.

5.2 Detecting Deadlocks and Livelocks

Consider the circuit shown in figure 4. The components used in this circuit, before the step of adding *failure paths*, have the following behaviors:

```
Gselector = ain.('bout.Gselector+'cout.Gselector)
Merge     = a.'c.Merge + b.'c.Merge
```

We connect `bout` to the external output `b`, `cout` to `x2`, the `b` input of `Merge` to `x2`, the `c` output of `Merge` to `x1`, and the `ain` input of `Gselector` to `x1`. Then, after applying a transition at the `A` input, the circuit can engage in a sequence of `X1,X2,X1,...` actions of arbitrary length before it emits a `B` (depending upon the "fairness" of selection of unit `Gselector` (shown as G.S. in the Figure). Dill's verifier is incapable of pointing out that the circuit can diverge; the CWB is able to do this. If we now consider the circuit shown in figure 5: (1) the conformance check passes the deadlockable wire as a safe substitution for a wire; (2) the strong conformance check rightly points out that the deadlockable wire is *not* a safe substitution for a wire; and (3) the CWB is able to detect a deadlock.

6 Conclusions

We have identified some of the precautions necessary to be observed before CWB can be applied for verifying asynchronous circuits. We have also presented two approaches, the addition of a `Realwire` component at every modules' input, or alternately, the process of adding *failure paths*, to convert CCS specifications into those that exhibit all[1] *chokes* and *autofailures*.

By taking one of these approaches, the CCS/CWB combination becomes a powerful tool that is capable of detecting circuit errors, and also permit checking for divergences, deadlocks (even those other than the ones caused by `Chokes`), and also user-given modal properties. (Note: there is ongoing work at the Carnegie Mellon University to study the use of both trace theory and various temporal logics for asynchronous circuit verification.) Another advantage we see with the CCS/CWB approach is that it permits both high level protocols as well as low-level implementations of these protocols to be reasoned about using the same tool.

Currently the CWB is not very efficient – even moderately sized circuits take a long time to run. Dill's verifier, on the other hand, executes much faster. Perhaps the CWB can be re-coded to solve this.

In conclusion, we believe that we have identified some useful connections between Dill's trace theory and the CCS model from the point of view of asynchronous circuit verification.

Acknowledgements: Graham Birtwistle and his group at the University of Calgary taught me about CWB, and the use of CCS for asynchronous circuit verification, for which I am grateful. Thanks also to Steve Nowick, Nick Michell, and Erik Brunvand for valuable discussions.

References

[1] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

[2] John Aldwinckle, Rajagopal Nagarajan, and Graham Birtwistle. An introduction to modal logic and its applications on the concurrency workbench (preliminary version). Technical Report 92/467/05, University of Calgary, February 1992.

[3] Venkatesh Akella and Ganesh Gopalakrishnan. Static analysis techniques for the synthesis of efficient asynchronous circuits. Technical Report UUCS-91-018, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991. *To appear in TAU '92: 1992 Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, Princeton, NJ, March 18–20, 1992.*

[1] We are working on a formal argument to support this claim.

[4] Venkatesh Akella and Ganesh Gopalakrishnan. hopcp: A concurrent hardware description language. Technical Report UUCS-TR-91-021, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991.

[5] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

[6] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In editor C.A.R. Hoare, editor, UT Year of Programming Institute on Concurrent Programming. Addison-Wesley, 1989.

[7] C. H. van Berkel, C. Niessen, M. Rem, and R.W. J. J. Saeijs. VLSI programming and silicon compilation: a novel approach from Philips Research. In *Proc ICCD*, New York, 1988.

[8] Jo C. Ebergen. *Translating Programs into Delay Insensitive Circuits.* Centre for Mathematics and Computer Science, Amsterdam, 1989. *CWI Tract 56.*

[9] Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

[10] Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990. *Being revised based on comments from the acm Computing Surveys.*

[11] George G. Milne and Mauro Pezze. Typed circal: A high level framework for hardware verification. In *Proc. 1988 IFIP WG 10.2 International Working Conference on "The Fusion of Hardware Design and Verification", Univ. of Strathclyde, Glasgow, Scotland*, pages 115–136, July 1988.

[12] Robin Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1989.

[13] Wei Chen, Jan Tijmen Udding, and Tom Verhoeff. Networks of communicating processes and their (de-)composition. In Jan L.A. van de

Snepscheut, editor, *Springer Verlag Lecture Notes in Computer Science, No.375, Mathematics of Program Construction*, pages 174–197. Springer Verlag, 1989.

[14] Ganesh Gopalakrishnan, Nick Michell, Erik Brunvand, and Steven M. Nowick. A correctness criterion for asynchronous circuit verification and optimization. *IEEE Transactions on Computer-Aided Design*, 13(11):1309–1318, November 1994.

[15] Rance Cleveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics based tood for the verification of concurrent systems. Technical Report ECS-LFCS-89-83, Laboratory for Foundations of Computer Science, Univ of Edinburgh, August 1989.

[16] Ganesh Gopalakrishnan. Micropipeline wavefront arbiters using lockable c-elements. *IEEE Design and Test of Computers*, 11(4):55–64, 1994. Winter 1994 Issue.

Notations:

- It is assumed that the network $T_I \parallel \overline{T_S}$ is closed (each output of $\overline{T_S}$ matches an input of T_I, and vice-versa), and no two outputs are connected together.
- Define $T_0 = \overline{T_S}$ and $T_1 = T_I$.
- Define T_{01} = the set $\{T_0, T_1\}$.
- Define $\tilde{T}$ = if $(T = T_0)$ then T_1 else T_0.
- Define $next(s, x)$ to be the next state attained from state s upon processing input/output x.
- Initialize a global set of state pairs, $visited = \phi$.
- Call conforms-to-p(T_{01}, start-state-0, start-state-1).
- Report "success".

conforms-to-p(T_{01}, st_0, st_1) =
if $(st_0, st_1) \in visited$
  then return
  else
    $visited := visited \cup \{(st_0, st_1)\}$;
    for each $T \in T_{01}$
      for each enabled output x of T
        if x is enabled in $\tilde{T}$
          then conforms-to-p($T_{01}, next(st0, x), next(st1, x)$)
          else *ERROR (print failure trace and abort)*
        end if
      end for
    end for
end if
end conforms-to-p

Figure 1: Algorithm for Checking for Conformance

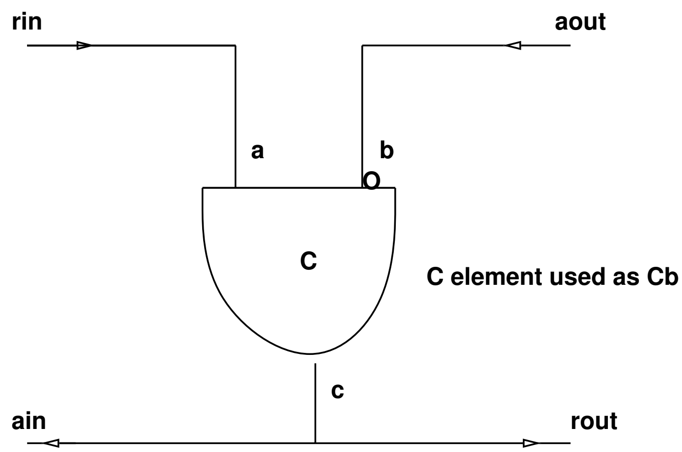

Figure 2: A C element used as a Micropipeline Stage

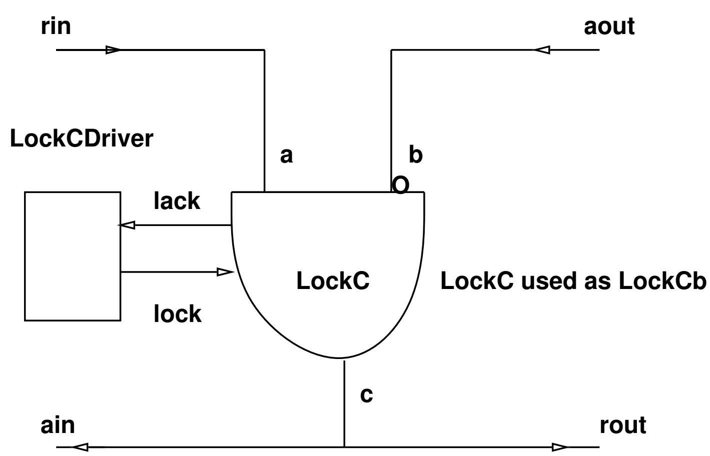

Figure 3: A LockC element used as a Micropipeline Stage

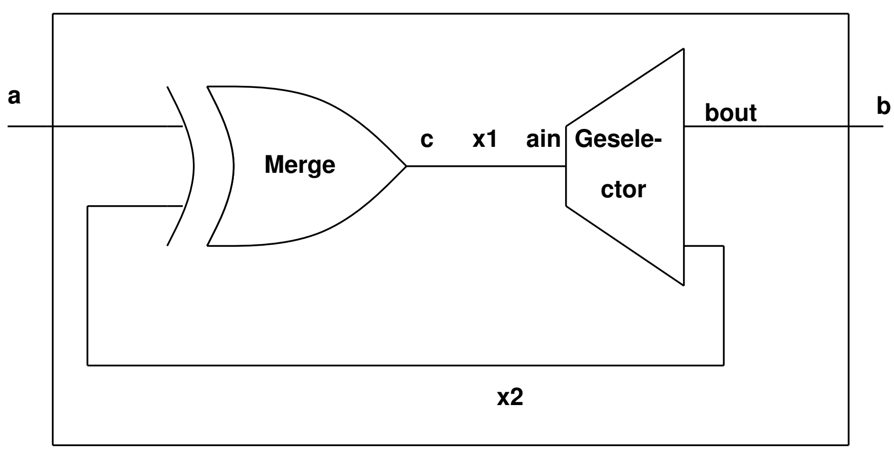

Figure 4: A Livelockable Wire

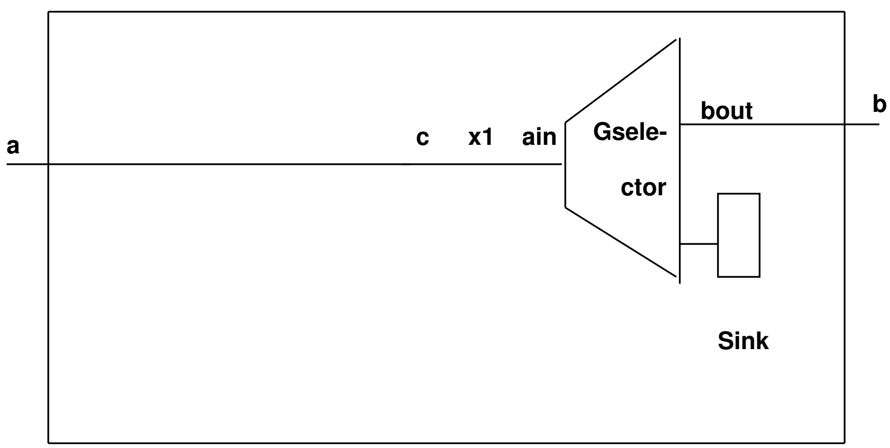

Figure 5: A Deadlockable Wire