A CORRECTNESS CRITERION FOR
ASYNCHRONOUS CIRCUIT VALIDATION AND OPTIMIZATION

GANESH GOPALAKRISHNAN1
ERIK BRUNVAND2
NICK MICHELL

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA
ganesh@cs.utah.edu, elb@cs.utah.edu, michell@cs.utah.edu

STEVEN M. NOWICK3

Computer Systems Laboratory,
Department of Computer Science,
Room #222 Margaret Jacks Hall,
Building 460,
Stanford University,
Stanford, CA 94305
nowick@cs.stanford.edu

UUCS-92-004

Abstract

We propose a new relation \( \subseteq \) called strong conformance in the context of Dill's trace theory, and define \( B \subseteq A \) to be true exactly when \( B \) conforms to \( A \) and the success set of \( B \) contains the success set of \( A \). When \( B \subseteq A \), module \( B \) operated in module \( A \)'s maximal environment \( A^M \) (i.e. \( B \parallel A^M \)) exhibits all the traces that \( A \parallel A^M \) exhibits. In addition, if \( A \) has a success trace \( x \), \( B \) can have additional success traces of the form \( xia^* \) where \( i \) is an input and \( a \) is the alphabet of the trace structure. This means that \( B \) can have additional capabilities that \( A \) does not. We show that strong conformance is more useful than conformance (defined by Dill) in detecting certain errors in asynchronous circuits. Strong conformance also helps justify circuit optimization rules that replace a component \( A \) by another component \( B \) that may have extra capabilities (e.g. can accept more inputs). The structural operators compose, rename, and hide of Dill's trace theory are monotonic with respect to strong conformance. Experiments using a modified version of Dill's trace theory verifier are presented.

Submitted to the 1992 Computer-Aided Verification Workshop and IEEE Transactions on CAD

---

1Supported in part by NSF Award MIP-8902558
2Supported in part by NSF Award MIP-9111793
3Supported in part by the Semiconductor Research Corporation, Contract no. 91-DJ-205, and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.
A Correctness Criterion for Asynchronous Circuit Validation and Optimization

GANESH GOPALAKRISHNAN*
ERIK BRUNVAND¹
NICK MICHELL
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

STEVEN M. NOWICK³
Computer Systems Laboratory,
Department of Computer Science,
Room #222 Margaret Jacks Hall,
Building 460,
Stanford University,
Stanford, CA 94305

Keywords: Asynchronous Circuits, Circuit Optimizations, Formal Verification of Hardware, Trace Theory

Abstract. We propose a new relation \( C \subseteq A \) called strong conformance in the context of Dill's trace theory [1], and define \( B \subseteq A \) to be true exactly when \( B \) conforms to \( A \) and the success set of \( B \) contains the success set of \( A \). When \( B \subseteq A \), module \( B \) operated in module \( A \)'s maximal environment \( A^M \) (i.e. \( B \parallel A^M \)) exhibits all the traces that \( A \parallel A^M \) exhibits. In addition, if \( A \) has a success trace \( x \), \( B \) can have additional success traces of the form \( x \alpha^* \) where \( i \) is an input and \( \alpha \) is the alphabet of the trace structure. This means that \( B \) can have additional capabilities that \( A \) does not. We show that strong conformance is more useful than conformance (defined by Dill) in detecting certain errors in asynchronous circuits. Strong conformance also helps justify circuit optimization rules that replace a component \( A \) by another component \( B \) that may have extra capabilities (e.g. can accept more inputs). The structural operators compose, rename, and hide of Dill's trace theory are monopionic with respect to strong conformance. Experiments using a modified version of Dill's trace theory verifier are presented.

1 Introduction

Asynchronous circuits are enjoying a revival, as designers confront problems associated with the scale of modern VLSI [2]. Despite the many advantages they offer, the number of temporal concerns involved in verifying asynchronous circuits is, in general, very large. In practice, the task of verifying asynchronous circuits is greatly simplified through environmental assumptions (e.g. single input changes) or by relying on circuit properties (e.g. delay insensitivity, which means that the circuit behavior is independent of delays on wires that lead to its terminals, or speed independence, which means that the circuit behavior is independent of the delays of its constituent gates).

Dill [1] has developed a trace theory and a verifier based on it; the verifier has been applied to a

*Supported in part by NSF Award MIP-8902558
¹Supported in part by NSF Award MIP-9111793
³Supported in part by the Semiconductor Research Corporation, Contract no. 91-DJ-205, and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

number of finite-state speed independent asynchronous circuits and has uncovered bugs in several published circuits. Nowick [3] has integrated this verifier into the asynchronous circuit synthesis framework used by a research division of Hewlett-Packard. Despite the impressive performance of the verifier, the verification criteria it uses, namely conformance and conformation equivalence, are inadequate to detect many errors that can be introduced during speed independent circuit design or during circuit optimizations. In this paper, we propose a simple extension to conformance, called strong conformance, and point out when this criterion is useful and interesting during speed independent circuit verification. We first motivate the need for this notion through some examples. Then, we present the theoretical aspects of strong conformance. Finally, we present experiments that illustrate the strengths as well as the limitations of this notion.

There is also a more fundamental question: in what different ways can asynchronous circuits be compared, and when are they useful/interesting? This question arises quite naturally, because many comparison relations have been proposed in the area of process calculi such as CCS [4] and CSP [5] (for example, see [6]). Although we do not offer an answer to the above question, our work (in proposing strong conformance) can be seen as a step in this direction.

Our work was principally motivated by our inability to reason about the correctness of some of the optimization rules used in Brunvand's asynchronous circuit compiler [7, 8] using ideas that we were familiar with.

Section 2 presents the required background of Dill's trace theory, and defines conformance, which is the comparison relation used by Dill. It also presents the notion of conformation equivalence. Section 3 defines strong conformance as a small extension to conformance, and presents an algorithm for verifying this new relation. The central idea underlying this paper can be illustrated through a few examples, which are presented in section 4. In presenting examples, we shall use a simple notation to describe FSMs. Section 5 examines the properties of strong conformance. Section 6 presents experiments with an implementation of strong conformance in Dill's trace theory verifier. Section 7 discusses our results, related work, and provides concluding remarks. The Appendix provides deferred details.

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. In the following discussion, we assume that \(S\) is a non-empty set.

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 \(\alpha\). The
failure set of a module is the set of traces that correspond to "improper uses" of the module. A failure set of a module is completely determined by the success set: \( F = (SI - S)a^* \). Intuitively, \((SI - S)\) describes all strings of the form \( xa \), where \( x \) is a success and \( a \) is an "illegal" input signal. 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, \( \epsilon \)), 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.

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:

\[
WIRE = a? \rightarrow b! \rightarrow WIRE
\]

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 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 \( T_S || T' \) has no failures, \( T_I || T' \) 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:

\[
J = a? \rightarrow b? \rightarrow c! \rightarrow J\quad | \quad b? \rightarrow a? \rightarrow c! \rightarrow J
\]
Now, consider a modified JOIN:

\[ J_1 = a? \rightarrow b? \rightarrow c! \rightarrow J \]

Notice that the success set of \( J_1 \) leaves out the trace \( b;a;c \). Clearly it is not safe to substitute \( J_1 \) for \( J \): \( J_1 \) 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 \( J_1 \not\leq J \), since the implementation cannot accept an input transition which the specification can receive.

However, note that it is safe to substitute \( J \) for \( J_1 \), since \( J \) can handle every input (and more) which \( J_1 \) can handle; so \( J \leq J_1 \). 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:

\[
\text{CONCUR.MOD} = a? \rightarrow (b'! \parallel c'!) \rightarrow \text{CONCUR.MOD} \\
\text{SEQNTL.MOD} = a? \rightarrow b'! \rightarrow c'! \rightarrow \text{SEQNTL.MOD}
\]

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

However, \( \text{SEQNTL.MOD} \) can be safely substituted for \( \text{CONCUR.MOD} \) in any environment. Any environment accepting outputs from \( \text{CONCUR.MOD} \) will also accept outputs generated by \( \text{SEQNTL.MOD} \), so \( \text{SEQNTL.MOD} \leq \text{CONCUR.MOD} \). 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:

\[
\text{AlmostWood} = a? \rightarrow b? \rightarrow c! \rightarrow \text{AlmostWood} \\
| b? \rightarrow a? \rightarrow \text{AlmostWood}
\]

The reason why \( J \) can be safely substituted by \( \text{AlmostWood} \) in any context is the following. So long as the environment and the component keep generating the sequence \( abcabcabc \ldots \), both \( J \) and \( \text{AlmostWood} \) 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; \( \text{AlmostWood} \), 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

\[
\text{BlockOfWood} = a? \rightarrow \text{BlockOfWood} \\
| b? \rightarrow \text{BlockOfWood}
\]

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.

2.3 On Establishing Conformance

A verifier has been developed by Dill to establish conformance. Relation \( \leq \) is established in this verifier as follows (we use \( T, T_s, \) etc. to denote trace structures):

- The verifier constructs a trace structure, \( T_s \), called the mirror of specification \( T_s \) (see [1]; originally proposed in [9]). \( T_s \) 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, \( T_s \parallel T_s \). It has been proven that \( T_i \leq T_s \) iff \( T_i \parallel T_s \) is failure-free (see [1]).
- \( T_i \leq T_s \) is checked by testing that \( T_i \parallel T_s \) 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 \( \bar{J} \), where \( \bar{J} \) is the mirror of specification \( J \):

\[
\begin{align*}
\bar{J} &= a! \rightarrow b! \rightarrow c? \rightarrow \bar{J} \\
&| b! \rightarrow a! \rightarrow c? \rightarrow \bar{J}
\end{align*}
\]

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

2.4 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 (\( A \equiv B \)) if \( A \leq B \) and \( B \leq A \) (see [1]).

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

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

How do we rescue the situation? The answer lies in not attempting \( \text{Spec} \leq \text{Imp} \), but merely whether \( S_{\text{Spec}} \subseteq S_{\text{Imp}} \), where '\( S_{M} \)' denotes the success set of '\( M \)'. This is the strong conformance relation that we have defined.
Notations:

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

```plaintext
conforms-to-p(T_{0i}, s_0, s_1) =
if (s_0, s_1) \in \text{visited}
then return
else
    \text{visited} := \text{visited} \cup \{(s_0, s_1)\};
    for each $T \in T_{01}$
        for each enabled output $x$ of $T$
            if $x$ is enabled in $\overline{T_f}$
                then conform-to-p(T_{0i}, next(s_0, x), next(s_1, x))
            else ERROR (print failure trace and abort)
        end if
    end for
end if
end conforms-to-p
```

Figure 1: Algorithm for Checking for Conformance

3 Strong Conformance

Definition: We define $T \subseteq T'$, read $T$ conforms strongly to $T'$, if $T \leq T'$ and $S_T \supseteq S_{T'}$. The algorithm to check for strong conformance is presented in Figure 2.

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.

4 Examples Illustrating Strong Conformance

Example 1

Consider a specification for an asynchronous circuit to be built, given in a state-transition oriented notation:

$$Spec = a? \rightarrow a'! \rightarrow Spec$$
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

Notations: Same as in Figure 1.

- Initialize a global set of state pairs, \( \text{visited} = \emptyset \).
- Call strong-conforms-to-p\((T_0, \text{start-state-0}, \text{start-state-1})\).
- Report "success".

\[
\text{strong-conforms-to-p}(T_0, \text{start-state-0}, \text{start-state-1}) = \\
\begin{cases}
\text{if } (\text{start-state-0, start-state-1}) \in \text{visited} & \text{then return} \\
\text{else } \text{visited} := \text{visited} \cup \{(\text{start-state-0, start-state-1})\} \\
& \text{for each enabled input } x \text{ of } T_0 \quad (* \text{Strong conformance checking loop } *) \\
& \quad \text{if } x \text{ is not enabled in } T_1 \\
& \quad \quad \text{then } \text{ERROR (print failure trace and abort);} \\
& \quad \text{end if} \\
& \text{end for} \\
& \text{for each } T \in T_0 \\
& \quad \text{for each enabled output } x \text{ of } T \\
& \quad \quad \text{if } x \text{ is enabled in } T \\
& \quad \quad \quad \text{then } \text{strong-conforms-to-p}(T_0, \text{next}(\text{start-state-0}, x), \text{next}(\text{start-state-1}, x)) \\
& \quad \quad \quad \text{else } \text{ERROR (print failure trace and abort)} \\
& \quad \quad \text{end if} \\
& \text{end for} \\
& \text{end if} \\
\end{cases}
\]

Figure 2: Algorithm for Checking for Strong Conformance

\( | b? \rightarrow b!' \rightarrow \text{Spec} \)

This specification describes a component having input terminals \( a \) and \( b \), output terminals \( a' \) and \( b' \), and the behavior of process \( \text{Spec} \). Process \( \text{Spec} \) awaits signal transitions on both terminals \( a \) and \( b \). If the first transition occurs on input terminal \( a \), it generates an output transition on terminal \( a' \), and continues to behave as process \( \text{Spec} \); If the first transition occurs on terminal \( b \), it generates an output transition on terminal \( b' \) and similarly continues to behave as process \( \text{Spec} \).

The behavior of \( \text{Spec} \) can be realized in many ways. Strangely enough, it is cheaper (i.e. takes fewer components—in fact none!) to overimplement \( \text{Spec} \) than to implement exactly the required behavior. The simplest implementation of \( \text{Spec} \) consists of just two WIREs. These WIREs are used to connect input \( a \) directly to output \( a' \), and input \( b \) directly to output \( b' \). This is an overimplementation, as the state-transition specification starting at process \( \text{TwoWires} \) shows:

\[
\text{TwoWires} = a? \rightarrow \text{AFull} \\
| b? \rightarrow \text{BFull} \\
\text{AFull} = b? \rightarrow \text{ABFull}
\]
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

| a'! → TwoWires

\[ ABFull = a'! → BFull \]
| | b'! → AFull

\[ BFull = a? → ABFull \]
| | b'! → TwoWires

The implementation TwoWires is an over-implementation, because it can accept more input sequences than required; for example, one \( a \) followed by one \( b \). However it is a correct implementation, because it supports all the behaviors that Spec supports, and therefore can be safely substituted for Spec in any context.

Notice that TwoWires \( \triangleright_{\text{conf}} \) Spec. However, TwoWires \( \preceq \) Spec.

We also have TwoWires \( \subset \) Spec. Superficially, it may seem that \( \preceq \) and \( \subset \) are the same – but the examples to follow show that this is not the case.

Example 2

Consider the specification of the Universal do nothing module BlockOfWood, described earlier [1]:

\[ BlockOfWood = a? → BlockOfWood \]
| | b? → BlockOfWood

Now consider the specification of a JOIN element:

\[ J = a? → b? → c! → J \]
| | b? → a? → c! → J

According to Dill’s trace theory, BlockOfWood conforms to \( J \); and therefore \( J \) may be substituted by BlockOfWood. However, BlockOfWood deadlocks and is therefore an undesirable substitution. The check BlockOfWood \( \subseteq J \) fails, and on this basis we can reject BlockOfWood as a replacement for \( J \). In this example, for our purposes, \( \subseteq \) is superior to \( \preceq \).

Example 3

Consider the following circuit built using a generalized selector GS:

\[ GS = a? → (b! → GS | c! → GS) \]

where \( | \) denotes choice (in this example, a non-deterministic choice). We now build three versions of WIREs using GS (the examples were provided by Ebergen [10]) shown in Figure 3. These examples illustrate some of the limitations of a trace theory of simple prefix closed trace structures when it comes to modeling liveness properties. Three examples clarify these remarks:

DL-WIRE can deadlock after some \( a \)'s and \( b \)'s, if GS “makes the wrong choice”. LL-WIRE, after receiving an \( a \), can livelock, engaging in an arbitrarily long \( (X1 \times X2) \) trace without emitting a ‘b’.
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

The verifier based on prefix-closed trace structures ignores these possibilities and constructs state transition diagrams for DL-WIRE and LL-WIRE that match the state transition diagram of WIRE realized using a buffer. Thus, both conformance and strong conformance fail to detect errors in DL-WIRE and LL-WIRE.

This example illustrates cases of deadlock and livelock which cannot be modeled by strong conformance. Though the theory of complete trace structures [1, Chapter 7] holds promise in being able to distinguish between WIRE and DL-WIRE/LL-WIRE, tool support does not currently exist for checking for conformance in the realm of complete trace structures. This will be a subject for future research.

Example 4

Consider the specification of an alternating selector [1]:

\[ AS = a? \rightarrow b! \rightarrow a? \rightarrow c! \rightarrow AS \]

AS \preceq GS (but not vice-versa) showing that AS is a safe substitution for GS. However, neither AS \sqsubseteq GS (because \(S_{AS}\) does not contain \(S_{GS}\) – in fact, \(S_{GS}\) contains \(S_{AS}\)! nor GS \sqsubseteq AS (because GS does not conform to AS).

It may be argued that in some situations, we can accept an AS as a legal substitution for GS (e.g. if a two-way round-robin scheduler is acceptable in place of a two-way random scheduler). However, because such a replacement will reduce the number of traces that can be observed from AS || GS in comparison with GS || GS\(^M\) (i.e. only strictly alternating as and bs are allowed), we can justify the failure of the check AS \sqsubseteq GS. Thus, \sqsubseteq may occasionally “err”, discarding possibly useful implementations, because of its strict definition of acceptable output behavior.

5 Properties of the Strong Conformance Relation

It can be shown that strong conformance is transitive (because \preceq and \subseteq are transitive), and that the structural operators compose, rename, and hide are monotonic with respect to strong conformance (because they are monotonic with respect to \preceq as shown in [1]). Thus, replacing a component A by another component B in a system S[], where B \subseteq A, ensures that S[B] \subseteq S[A] – or the new system is no worse.

We also have the following result.
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

Proposition. If \( B \subseteq A \), then \( S_{(A \mid| A^M)} = S_{(B \mid| A^M)} \). In other words, if \( B \subseteq A \), the "simulation" of \( A \) with its maximal environment \( A^M \) will exhibit the same success traces as the simulation of \( B \) with \( A^M \). (Note: The simulation of \( A \) with \( A^M \) is guaranteed to be free of failures, by the definition of mirror. Since \( A \) and \( B \) are canonical ([1]), their success and failure sets are disjoint. Therefore, since \( S_B \supseteq S_A \), the simulation of \( (B \mid| A^M) \) will also be free of failures. Thus, the only thing we can compare are the success sets.)

Viewed yet another way, \( B \) can be replaced for \( A \) in any environment, up to the maximal environment \( A^M \), and one will not observe any difference in the set of transactions that cross the boundary between \( A^M \) and \( A \) or \( A^M \) and \( B \). 

Proof Outline. The proof is by induction over the length of traces. As basis case, \( \epsilon \) is in the success sets of \( (A \mid A^M) \) and \( (B \mid A^M) \). Now assume that a trace \( x \) of length \( N \) is in both success sets. Consider a trace \( x_i \) in the success set of \( (A \mid A^M) \), in other words, \( A^M \) sends an output \( i_1 \) to \( A \) that \( A \) can accept; then, the same input \( i_1 \) must be accepted by \( B \) also, because \( S_B \supseteq S_A \), and so \( x_i \) is in the success set of \( (B \mid A^M) \) also. By the same argument, if \( x_{\alpha_1} \) is in \( S_{A\mid A^M} \), it will be in \( S_A \), which means it is in \( S_B \), which means it is in \( S_{B\mid A^M} \) also. If there is an \( i_2 \) such that \( x_{i_2} \) is in \( S_B \) but not in \( S_A \), then \( x_{i_2} \) is not in \( (A \mid A^M) \) or in \( (B \mid A^M) \), because \( A^M \) will not generate the output \( i_2 \) — the "excess capability" of \( B \) to accept \( i_2 \) remains unused (and, hence, \( i_2 \) can lead to any behavior in \( B \), including \( \alpha^* \)). Finally, if \( x_{i_2} \) is in \( S_B \) but not in \( S_A \), we have a contradiction, because \( B \subseteq A \) by definition implies \( B \subseteq A \), and a module (like \( B \)) that conforms to another (like \( A \)) cannot generate an "excess output" \( \alpha_2 \) that \( A \) cannot generate.

This proof exactly characterizes the notion of strong conformance: \( A \) conforms strongly to \( B \) if it offers to accept excess inputs at certain states that \( A \) doesn’t offer to accept. This is harmless, because the maximal environment of \( A \) where \( B \) will be plugged in as a replacement will not draw upon these excess capabilities of \( B \).

6 Experimental Results

6.1 Call-Merge Optimization

The initial circuits generated by either the occam [7] or the hopCP [11, 12] synthesis system have a number of redundancies. These redundancies arise, because the HDL constructs are compiled without taking their contexts into account. While the circuit is being optimized, certain circuits get so constrained in their usage that they can be replaced with other (cheaper) circuits, as shown in [7]. An example of this, from [7], is shown in Figure 4.

Suppose that a circuit contains the CALL element shown to the left. The behavior of CALL is

\[
\text{CALL} = a? \rightarrow c! \rightarrow c? \rightarrow a! \rightarrow \text{CALL}
\]
Figure 5: Petri Net Specification of a Queue

\[ | b? \rightarrow c'! \rightarrow c? \rightarrow b'! \rightarrow CALL \]

Suppose that during the course of optimization, the \( c' \) output of \( CALL \) gets connected back to its \( c \) input as shown in \( CALLX \). It is assumed that \( CALL1 \) is being operated in a delay insensitive context, as the original circuit was. The delay insensitized behavior of \( CALL1 \) is

\[
\begin{align*}
CALL1 &= a? \rightarrow (c'! || a'!) \rightarrow CALL1 \\
&| b? \rightarrow (c'! || b'!) \rightarrow CALL1
\end{align*}
\]

where the notation means: after performing \( a? \), perform \( c'! \) and \( a'! \) in some order before repeating the behavior of \( CALL1 \) (and similarly for the second branch of the choice). This circuit can be replaced by \( MCALLX \) which is smaller and faster than \( CALLX \). Clearly \( MCALLX \) is not equivalent to \( CALL1 \), because the execution sequence

\( a?; c'!; b? \)

is possible for \( MCALL1 \) but not for \( CALL1 \).

We have \( MCALL1 \preceq CALL1 \) as well as \( MCALL1 \preceq CALL1 \). The latter check assures us that \( MCALL1 \) exhibits all the successful traces of \( CALL1 \).

### 6.2 Error Detection in Queue Cell

Consider a queue cell \( CONCUR-Q \) specified in Figure 5, where the capacity is set to 1. The queue cell can be realized using the familiar micropipeline circuit \( QIMP1 \) shown in Figure 6:

Suppose the circuit is implemented by mistake as \( QIMP2 \). Though \( QIMP2 \) conforms to \( CONCUR-Q \), \( QIMP2 \) does not conform strongly to \( CONCUR-Q \). This “implementation” does nothing wrong, but deadlocks immediately. The strong conformance check fails, and generates the error message:
6.3 A 4-phase to 2-phase Converter with Quick Return

Consider the specification of a four-phase to two-phase converter with "quick return":

\[ QR42 - SPEC = r4? \rightarrow ((a4! \rightarrow r4?) \parallel (r2! \rightarrow a2?)) \rightarrow a4! \rightarrow QR42 - SPEC \]

where \(((a \rightarrow b) \parallel (c \rightarrow d))\) represents all possible overlapped executions of \((a \rightarrow b)\) and \((c \rightarrow d)\).

We obtain that QR42-IMP conforms to QR42-SPEC. However QR42-IMP does not conform strongly to QR42-SPEC, because QR42-SPEC allows the trace \((R4 A4)\) while QR42-IMP does...
not. QR42-SPEC is an abstract specification that allows R2 as well as A4 to occur concurrently immediately after R4. This means that the environment of QR42-SPEC is free to accept R2 and A4 in any order. The circuit QR42-IMP, however, generates A4 only following R2, thus not invoking certain capabilities that exist in the environment.

Notice that QR42-IMP does not obey the delay insensitive signaling protocol: it allows the trace (R4 R2 A4) but not (R4 A4 R2); however, if a buffer is introduced on the r2 wire, the sequence (R4 A4 R2) will also become possible. Thus, a delay insensitized version of QR42-IMP conforms strongly to QR42-SPEC.

6.4 31-Location Queue in Place of a 32-Location Queue

Finally, we experimented with a 31-location queue in place of a 32 location queue. Conformance passed the 31-location implementation, since the 31-location queue can be safely substituted for the 32-location queue. However, this implementation certainly has more limited output behavior than the specification. On the other hand, the strong conformance check detects this limited output behavior; it finds the following sequence leading to an error:

```
(STRONG-CONFORMS-TO-P *concur-Q31* *concur-Q32*)
...
Failure path: (RIN AIN RIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN RIN AIN AIN RIN AIN AIN RIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN RIN AIN AIN AIN)
```

The strong conformance checker could find the failure in 0.1 seconds and print out a sequence of actions leading to the error, despite having 128 states in the specification. This means that the trace theory based verifier does not always suffer from combinatorial explosion, as is commonly feared. (However, in large regular array structures where each cell has a lot of independent moves possible, the explosion could well manifest – as communicated to us by Birtwistle in the context of the Concurrency Workbench. See [13] for further discussion.)

7 Discussion, Related Work, and Conclusions

A relation strong conformance between trace structures has been presented and its various uses have been pointed out. This notion is closely related to the definition of decomposition presented in [9]. Key differences are briefly noted. Ebergen’s trace theory is designed with different objectives: to specify computations, and synthesize circuits through calculations using trace-theoretic rules. The trace theory does not directly relate to circuit components; for instance, two trace structures containing the same output symbol can be weaved. Weave merely captures constraints on joint execution, and does not correspond to the act of connecting two circuit outputs (which Dill’s || operator has, and hence Dill prevents composing two trace structures containing the same output symbol).

In Ebergen’s trace theory, the link between trace theoretic operators and circuit behavior is brought out through the following key notions and theorems: decomposition, Dl decomposition, separation theorem, and substitution theorem. Together with a rich collection of equational laws on commands (where commands denote trace structures), Ebergen’s trace theory seems capable of synthesizing correct circuits, without having to first “guess” a circuit and “check” it using a verifier (as has been
A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION

the approach suggested here). A tool to demonstrate the power of Ebergen's trace theory is under development. These works address the two prevalent points of view: post-hoc verification after "intelligent human design" vs. "correct by construction" design.

The notion of strong conformance is latent in Ebergen's definition of the decomposition relation [9, Definition 3.1.0.0, Page 42] – as was discovered after the fact by us. A very similar idea called input liberalization has also been proposed by Ad Peeters [14] – again discovered after the fact by us! Neither Ebergen nor Peeters suggest using their definitions for validating circuit optimizations, as we do here.

The process algebra developed by Udding and Josephs holds promise to contain state explosion [15, Remark on page 2], as circuits are derived through calculations in their process algebra, rather than verified post-hoc using a semantic model (state graphs) as with Dill's verifier. However, so long as the two points of view – post-hoc verification after "intelligent human design" vs. "correct by construction" design using intelligent calculations – exist, both approaches have an important role to play.

Finally, work in verification of asynchronous circuits seems to be proceeding along (at least) two distinct lines: (1) a class of works that use various trace models; (2) a class of works based on process algebras. Many of the notions used in these areas seem to be so conceptually similar (e.g. compare autofailure manifestation which converts possible failures to actual failures, and may/must pre-orders used by [6]). However there are fundamental differences in these approaches also (e.g. unidirectional wires carry information only one-way; so a component cannot refuse an input; however, a CCS/CSP rendezvous can be refused by not participating in it). One hopes to see unifying efforts relating these (as yet unrelated) efforts.

Acknowledgements. Thanks to Jo Ebergen for his insightful feedback on an earlier version of this paper.

References


A CORRECTNESS CRITERION FOR ASYNCHRONOUS CIRCUIT VALIDATION


8 Appendix

**Proposition.** *compose, rename, and hide* are monotonic with respect to strong conformance.

**Proof Outline.** These structural operators are monotonic with respect to \(\preceq\) as shown in [1, Page 58]. We are now required to show the additional facts that \(S_B \supseteq S_A\) implies

\[
\begin{align*}
S_{\text{hide}(X)(B)} & \supseteq S_{\text{hide}(X)(A)} \\
S_{\text{rename}(r)(B)} & \supseteq S_{\text{rename}(r)(A)} \\
S_B|_C & \supseteq S_A|_C
\end{align*}
\]

Equation 1 follows from the fact that \(\text{hide}(X)\) is a function that simply removes members of \(X\) from every success trace in \(S_A\) or \(S_B\) (as the case may be). Equation 2 follows from the fact that \(\text{rename}(r)\) simply applies the renaming function \(r\) to every in \(S_A\) or \(S_B\) (as the case may be). Finally, equation 3 follows from the fact that \(S_B|_C = S_B \cap S_C\) and \(S_A|_C = S_A \cap S_C\).