A Correctness Criterion for Asynchronous Circuit Validation and Optimization

GANESH GOPALAKRISHNAN* (ganesh@cs.utah.edu)
ERIK BRUNVAND† (brunvand@cs.utah.edu)
NICK MICHELL (michell@cs.utah.edu)

University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

STEVEN M. NOWICK‡ (nowick@cs.columbia.edu)

Department of Computer Science
Room 508, Computer Science Building
Columbia University
New York, New York 10027

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

Abstract. *In order to reason about the correctness of asynchronous circuit implementations and specifications, Dill has developed a variant of trace theory [1]. Trace theory describes the behavior of an asynchronous circuit by representing its possible executions as strings called "traces". A useful relation defined in this theory is called* conformance, *which holds when one trace specification can be safely substituted for another. We propose a new relation in the context of Dill's trace theory, called* strong conformance. *We show that this relation is capable of detecting certain errors in asynchronous circuits that cannot be detected through conformance. Strong conformance also helps to justify circuit optimization rules where a component is replaced by another component having extra capabilities* (e.g., *it can accept more inputs). The structural operators of Dill's trace theory — compose, rename and hide — are shown to be monotonic with respect to strong conformance. Experiments are presented using a modified version of Dill's trace theory verifier which implements the check for strong conformance.*

1 Introduction

Asynchronous circuits are enjoying a revival, as designers confront problems associated with the complexity of modern VLSI circuits [2]. Despite their many potential advantages, however, the verification of asynchronous circuits remains a difficult problem. Asynchronous circuits have been designed assuming a wide variety of delay models for gates and wires [3, 4]. Furthermore, a number of environmental modes have been used to define a circuit's interaction with its environment, such as *fundamental* [15] and *input/output* modes [7]. In practice, the task of verifying asynchronous circuits is greatly simplified by considering only particular classes of behavior, *e.g., delay-insensitivity* [31], where a circuit's correct operation is independent of delays in circuit components and in the wires

*Supported in part by NSF Award MIP-8902558

†Supported in part by NSF Award MIP-9111793

‡Supported in part by the Semiconductor Research Corporation, Contract nos. 91-DJ-205 and 92-DJ-205, and by the Stanford Center for Integrated Systems, Research Thrust in Synthesis and Verification of Multi-Module Systems.

that connect them; or *speed-independence* [6], where a circuit's correct operation is independent of delays in components, while wires are assumed to have negligible delay.

Dill [1] has developed a trace theory for the specification and verification of asynchronous circuits. Trace theory uses the theory of regular languages to model asynchronous circuits by representing executions as strings called "traces." The symbols in these traces represent signal transitions on the interface terminals of the circuit being represented. Dill has also developed a verifier based on trace theory. The verifier has been applied to a number of speed-independent asynchronous circuits [8, 5] and has uncovered bugs in several published circuits [1]. Nowick [9] has integrated this verifier into the asynchronous circuit synthesis framework used by a research division of Hewlett-Packard [10, 11]. Despite the impressive performance of the verifier, the verification criteria it uses, namely *conformance*, is inadequate to detect certain classes of commonly occuring errors that can be introduced during speed-independent and delay-insensitive circuit design or during circuit optimization. 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 and delay-insensitive 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.

Our work on verification raises a fundamental question: what are the most appropriate ways to compare asynchronous circuits, and when are the different approaches useful? This question arises quite naturally, because many comparison relations have been proposed in the area of process calculi such as CCS [12] and CSP [16] (for example, see [17]). Although we do not offer a definitive answer to this question, strong conformance can be seen as one useful contribution to the practical verification of asynchronous circuits.

This 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 [18, 19] using existing verification methods.

Section 2 presents the required background of Dill's trace theory, and defines conformance, which is the comparison relation used by Dill. Section 3 defines strong conformance as a small extension to conformance. First, we present an algorithm for verifying this new relation. Next, we provide two examples illustrating strong conformance. Finally, we examine the formal properties of strong conformance. Section 4 presents experiments with an implementation of strong conformance in Dill's trace theory verifier. Section 5 discusses results, related work and conclusions.

2 Background: Trace Theory

In the past decade or so, different *trace theories* have been developed by various researchers. These trace theories have been applied to the study of concurrent systems: by Hoare [16, Chapter 2], to the characterization of CSP processes; by Rem, Snepscheut, Udding [20, 21] and Ebergen [22] to the analysis, verification, and characterization of speed-independent and delay-insensitive circuits. This paper follows the version of trace theory proposed by Dill [1], who has applied his theory to the verification of speed-independent circuits. Dill has also extended his theory of *simple* *trace structures* to *complete trace structures* (which are capable of modeling infinite computations) mainly for the study of liveness properties. Because the operations and decision procedures for finite automata on infinite sequences are much more complicated [1], it is not clear how successful the practical adaptation of the theory of complete trace structures will be in the area of asynchronous circuit verification. (For a discussion of related issues, see [23, 24].)

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 the *alphabet* $\alpha = I \cup O$, 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 by 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. A 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. 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 (non-inverting) BUFFER with input a and output b. In this context, we view a buffer as a component that accepts signal transitions on a and produces signal transitions on b after an unspecified delay. If we were to use BUFFER properly, its successful executions would include one where it has done nothing (*i.e.*, has produced trace ϵ), one where it has accepted an a but has not yet produced a b (*i.e.*, the trace a), one where it has accepted an a and produced a b (*i.e.*, the trace ab), and so on. More formally, the success set of BUFFER is $\{\epsilon, a, ab, aba, \ldots\}$. This set is a record of all the partial histories (including the empty one, ϵ), of successful executions of BUFFER. An example of an improper usage of BUFFER—a *choke*—is the trace aa. Once input a has arrived, a second change in a is illegal since it may cause unpredictable output behavior. A buffer of this type can be used to model a wire with some delay. Therefore, to transform a speed-independent circuit into a delay-insensitive circuit in the context of Dill's trace theory, buffers are attached to the terminals of the circuit.

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

$$\text{BUFFER} \quad = \quad a? \to b! \to \text{BUFFER}$$

In a process description, we use '$|$' to denote *choice*, '$\to$' 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). The above specification of BUFFER corresponds to the finite automaton in Figure 1 (which also shows the choke of BUFFER):

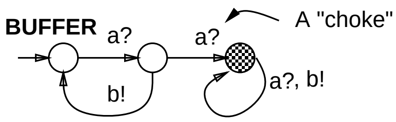

Figure 1: The Finite Automaton corresponding to BUFFER

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 omitted.

2.2 Operations on Trace Structures

There are two fundamental operations on trace structures: *compose* ($\|$) finds the concurrent behavior of two circuits that have some wires connected, and *hide* makes some output wires unobservable (suppressing irrelevant details of a circuit's operation). A third operation, *rename*, allows the user to generate modules from templates by renaming wires.

We consider the compose operation in more detail below (for further discussion, see [1]). The compose operator models the effect of connecting identically named wires between two circuits, called *components*. Given two components, A and B, with respective trace structures $T_A = (I_A, O_A, S_A)$ and $T_B = (I_B, O_B, S_B)$, the joint behavior of A and B is denoted by the trace structure $T_A \parallel T_B$. Components A and B can be composed only if they have no output wires in common, *i.e.,* $O_A \cap O_B = \emptyset$. If $T_{AB} = T_A \parallel T_B$, then the set of outputs of T_{AB} is $O_{AB} = O_A \cup O_B$ (whenever an output is connected to an input, the result is an output), and the set of inputs is $I_{AB} = (I_A \cup I_B) - O_{AB}$ (an input connected to an input is an input). Note that the alphabet, α_{AB}, of the composed trace structure is the union of the alphabets of the components, $\alpha_A \cup \alpha_B$.

The success set, S_{AB}, of T_{AB} is obtained from the success sets of T_A and T_B using a *product construction* method, sketched briefly below (for details, see [1]).

Product Construction Method to Define S_{AB}

As the success set for a component records the possible executions of the component, similarly the success set that records the possible *joint executions* of A and B, S_{AB}, must include only those executions that are "in agreement" with the executions of both A and B. The product construction method to define S_{AB} has two steps. **Step 1** determines those executions that are in agreement with the success sets of A and B; this step results in an intermediate success set S'_{AB}. **Step 2** then eliminates any "internal failures" that may be present in S'_{AB} (to be discussed below), to result in the final success set, S_{AB}. To help define S'_{AB}, we define $x \downarrow \alpha$ as the *projection* of trace x onto the alphabet α. The projection retains, in order, all the symbols in x that are also in α. For example, $abc \downarrow \{c, a\} = ac$, and $abc \downarrow \{d\} = \epsilon$.

Step 1: This step produces a set S'_{AB} of all traces, x, where the projection of x onto either alphabet, α_A or α_B, is a trace belonging to the corresponding success set, S_A or S_B. That is, actions on common symbols must occur through mutual consensus of the components, while actions on disjoint symbols (*i.e.,* symbols belonging to the alphabet of one component only) are governed only by the rules of operation of the corresponding component. Formally, set S'_{AB} contains all traces $x \in (\alpha_A \cup \alpha_B)^*$ where $(x \downarrow \alpha_A) \in S_A$ and $(x \downarrow \alpha_B) \in S_B$.

Step 2: This phase eliminates "internal failures" from S'_{AB} to obtain the final success set, S_{AB}. Consider a trace $x \in S'_{AB}$, which is a success in both components. Suppose that component A can successfully "extend" trace x by producing output a, where a then *causes a failure* in component B. In this case, once trace x has occurred, the composite circuit can cause its own failure, since component A may generate output a. As a result, to guarantee no failure in the composed circuit, *trace x itself must be avoided* — in effect, x must be classified directly as a failure. In general, a success trace x in the composed circuit is called an *autofailure* if x can be extended by one or more outputs to produce a failure in the composed circuit. The process of obtaining S_{AB} from S'_{AB} intuitively "exports" an internal failure to the interface of the circuit. That is, any input signal which ultimately causes a failure is considered as the direct cause of failure. Formally, we obtain S_{AB} from S'_{AB} as follows: Initially, let $S_{AB} = S'_{AB}$. For each trace $x \in S_{AB}$ and finite sequence of output symbols $y \in O^*_{AB}$, if $(xy \downarrow \alpha_A) \in F_A$ or $(xy \downarrow \alpha_B) \in F_B$, then $S_{AB} := S_{AB} - x$ (*i.e.,* remove x from S_{AB}). The resulting set S_{AB} is the final success set of $A \parallel B$.

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.)

Conformance holds when T_I can be *safely substituted* for T_S. More precisely, $T_I \preceq T_S$ if, for *every* context 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 could produce 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:

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

Next, consider a modified JOIN:

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

Notice that the success set of $J1$ omits the trace $b; a; c$. Clearly it is not safe to substitute $J1$ for J in all environments: $J1$ 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, it *is* safe to substitute J for $J1$, since J can handle every input (and more) that $J1$ can handle; so $J \preceq J1$. Thus, conformance 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:

$$\begin{aligned} CONCUR\_MOD &= a? \rightarrow (b'! \parallel c'!) \rightarrow CONCUR\_MOD \\ SEQNTL\_MOD &= a? \rightarrow b'! \rightarrow c'! \rightarrow SEQNTL\_MOD \end{aligned}$$

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 $SEQNTL\_MOD \preceq CONCUR\_MOD$. Thus, conformance allows an implementation to have "more constrained" output behavior than its specification.

This latter point can be illustrated more dramatically. We consider the earlier JOIN specification, J, and a new implementation:

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

J can be safely implemented by *AlmostWood* in any context for the following reason. As long as the component and its environment generate the sequence $abcabcabc\ldots$, J and *AlmostWood* behave alike. However, suppose that the environment generates the string ba and waits for output c. J will generate a c after seeing ba, thereby allowing the environment to proceed. *AlmostWood*, on the other hand, outputs nothing, and waits for a further a or b — at the same time as the environment is waiting for a c. In this case, the result is a deadlock. However, because no *incorrect* outputs are generated, *AlmostWood* is a safe substitution for J; that is, $AlmostWood \preceq J$.

Going to the extreme, consider the implementation:

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

This implementation also conforms to J: *BlockOfWood* does nothing useful, but neither does it cause any failures.

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, in practice, one often wants 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

As discussed earlier, in order to establish whether an implementation I conforms to a specification S (*i.e.*, $T_I \preceq T_S$), it is necessary in principle to show that I can be safely substituted for S in *all* contexts. Fortunately, a simpler method was first proposed by Ebergen [22] and further developed in the context of his work by Dill [1]. The *mirror*, $\overline{T_S}$, of S is defined as the trace structure whose input set is the output set of T_S, whose output set is the input set of T_S, and which has the same success set of T_S. Intuitively, the mirror is the worst-case environment which will "break" any trace structure that is not a true implementation of T_S.

More formally, given SPCTS T_I and T_S (with non-empty success sets), $T_I \preceq T_S$ *if and only if* $T_I \parallel \overline{T_S}$ is failure-free (*i.e.,* has an empty failure set). This result is proved and justified in [1]. Specifically, the mirror $\overline{T_S}$ produces as an output everything that T_S accepts as an input, so if T_I fails on any of these, there will be a failure in $T_I \parallel \overline{T_S}$. Similarly, $\overline{T_S}$ accepts as an input only what T_S produces as an output, so if T_I produces something else, there will be a failure in $T_I \parallel \overline{T_S}$ as well.

Using this result, Dill has developed a verifier to establish conformance. Given implementation I and specification S, with respective trace structures T_I and T_S, the verifier determines if $T_I \preceq T_S$ as follows:

1. Trace structures T_I and T_S are represented by deterministic finite automata [13].
2. Trace structure $\overline{T_S}$ is constructed.
3. The parallel composition, $T_I \parallel \overline{T_S}$, of implementation, T_I and mirror, $\overline{T_S}$, is obtained, using the product construction method described above.[1]
4. $T_I \preceq T_S$ is checked by determining whether $T_I \parallel \overline{T_S}$ is free of failures. This check is performed by searching the product automaton, depth-first, for a failure trace. If found, the failure trace is printed and the search is aborted.

Figure 2 presents the details of Step 4 of this algorithm.

[1] In practice, Dill's algorithm avoids the explicit construction of the product machine [1].

To illustrate the algorithm presented in Figure 2, we determine if the modified JOIN element, $J1$, conforms to the JOIN element, J, described earlier. The mirror, $\overline{J}$, of J is defined as follows:

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

We next obtain the composition $\overline{J} \parallel J1$ using the product construction method. Of the two components, $\overline{J}$ and $J1$, only $\overline{J}$ initially has an enabled output; in fact, both $a!$ and $b!$ are enabled in $\overline{J}$. While the production of $a!$ is acceptable for $J1$, the production of $b!$ by $\overline{J}$ will cause $J1$ to choke. Therefore, $J1 \not\preceq J$.

3 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 presented in Figure 3.

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

3.1 Examples Illustrating Strong Conformance

Example 1

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

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

This specification describes a component having input terminals a and b, output terminals a' and b', and the behavior of process $Spec$. Process $Spec$ waits for signal transitions on terminals a and b. If the first transition occurs on input terminal a, $Spec$ generates an output transition on terminal a', and continues to behave as process $Spec$. If the first transition occurs on terminal b, it generates an output transition on terminal b' and similarly continues to behave as process $Spec$.

The behavior of $Spec$ can be realized in many ways. One implementation consists of two (non-inverting) BUFFER components. In implementation $TwoWires$, the buffers are used to connect input a directly to output a', and input b directly to output b':

$$\begin{aligned} TwoWires &= WireA \parallel WireB \\ WireA &= a? \rightarrow a'! \rightarrow WireA \\ WireB &= b? \rightarrow b'! \rightarrow WireB \end{aligned}$$

$TwoWires$ is an "over-implementation" since it can accept more input sequences than required; for example, one a followed by one b. (Implementing exactly the required behavior, on the other hand, requires additional components.) However it is a *correct* implementation, because it supports all the behaviors that *Spec* supports. Therefore, *TwoWires* can be safely substituted for *Spec* in any context; that is, $TwoWires \preceq Spec$. Furthermore, *TwoWires* strongly conforms to *Spec* (i.e., $TwoWires \sqsubseteq Spec$). Superficially, it may seem that $\preceq$ and $\sqsubseteq$ are the same — but the following example shows that this is not the case.

Example 2

Consider the specification of the "universal do-nothing module" [1], *BlockOfWood*, described earlier:

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

Now consider the specification of a JOIN element:

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

According to Dill's trace theory, *BlockOfWood* conforms to J; therefore, *BlockOfWood* is a safe substitution for J. However, *BlockOfWood* deadlocks and is therefore an undesirable substitution. The strong conformance check $\mathit{BlockOfWood} \sqsubseteq J$ fails, and on this basis we can reject *BlockOfWood* as a replacement for J. In this example, for our purposes, $\sqsubseteq$ is superior to $\preceq$.

3.2 Properties of the *Strong Conformance* Relation

Strong conformance is a transitive relation, because $\preceq$ and $\subseteq$ are transitive. Other important properties of strong conformance are proved below.

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:

$$S_{hide(X)(B)} \supseteq S_{hide(X)(A)} \tag{1}$$

$$S_{rename(r)(B)} \supseteq S_{rename(r)(A)} \tag{2}$$

$$S_{B||C} \supseteq S_{A||C} \tag{3}$$

Equation 1 follows from the fact that $hide(X)$ is a function which 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 $rename(r)$ simply applies the renaming function r to every symbol 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 $S_{A||C} = S_A \cap S_C$. $\square$

In a practical sense, monotonicity is necessary for modular, or hierarchical, verification. For example, it would not help to show that $A \preceq B$ if this did not imply that for any context C, $(A \parallel C) \preceq (B \parallel C)$. More informally, we require of any practical system that if the replacement of a component is no worse than the replaced part, then the whole system is no worse after the substitution than before.

We also have the following result.

Proposition. If $B \sqsubseteq A$, then $S_{(A \| \overline{A})} = S_{(B \| \overline{A})}$. In other words, if $B \sqsubseteq A$, the composition of A with its maximal environment $\overline{A}$ (in the sense defined in Section 2.4) will exhibit the same success traces as the composition of B with $\overline{A}$.

Proof Outline. By definition, if $B \sqsubseteq A$, then $S_B \supseteq S_A$. Also, by definition, $S_{\overline{A}} = S_A$. Now, from the definition of $\|$, $S_{X \| Y} = S_X \cap S_Y$ if $\alpha_X = \alpha_Y$. Therefore, $S_{A \| \overline{A}} = S_A \cap S_{\overline{A}} = S_A = S_B \cap S_{\overline{A}} = S_{B \| \overline{A}}$. $\square$

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

This proof exactly characterizes the notion of strong conformance: B conforms strongly to A if B may offer to accept excess inputs in certain states where A cannot accept them. This excess capability of B is harmless, because the maximal environment of A cannot make use of this capability when B is used as a replacement for A.

4 Experimental Results

4.1 Error Detection in Queue Cell

A queue cell CONCUR-Q is specified by the Petri net [14, 8] in Figure 4, where the queue capacity is set to 1. The queue cell can be realized using the familiar micropipeline circuit $QIMP1$ shown in Figure 5.

Suppose that the circuit is erroneously implemented as $QIMP2$. $QIMP2$ is identical to $QIMP1$ except for a missing inversion bubble. (The $QIMP2$ description may be the result of a transcription or editing error, for example.) This "implementation" does nothing wrong, but deadlocks immediately.

$QIMP2$ conforms to CONCUR-Q, but $QIMP2$ does *not conform strongly* to CONCUR-Q. The strong conformance check fails, and generates the error message:

```
... failure trace (RIN AIN)
```

The trace indicates that the implementation cannot produce output `AIN` after receiving `RIN`, while CONCUR-Q can.

This example shows that strong conformance can detect certain forms of deadlock that are not detected by conformance. More precisely, if after seeing trace x, the specification has a successful extension through output o while the implementation does not, strong conformance fails.

4.2 1-Location Queue in Place of a 2-Location Queue

Next, we experiment with a 1-location queue used in place of a 2-location queue. Conformance passed the 1-location implementation, since the 1-location queue can be safely substituted for the 2-location queue. However, this implementation certainly has more limited output behavior than the specification. The strong conformance check detects this limited output behavior; it finds the following sequence leading to an error:

```
(STRONG-CONFORMS-TO-P *concur-Q1* *concur-Q2*)
...
Failure path: (RIN AIN RIN AIN)
```

The strong conformance check could find this failure almost immediately. Increasing the queue size did not increase the verification time substantially; for a 31-location queue in place of a 32-location queue, the error was detected after about 0.1 seconds on a 10-MIPS workstation.

4.3 Call-Merge Optimization

The initial circuits generated by either the occam [18] or hopCP [25, 26] synthesis systems have a number of redundancies. These redundancies arise because the HDL constructs are compiled without taking their contexts into account. During optimization, it is often possible to take advantage of a component's context, and thereby replace it with a cheaper component. An example of such an optimization, from [18], is shown in Figure 6.

Suppose that a circuit contains the $CALL$ element, shown in Figure 6. The behavior of $CALL$ is described as follows:

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

Suppose that during the course of optimization, the c' output of $CALL$ is connected back to its c input as shown in $CALL1$ in Figure 6. It is assumed that $CALL1$ is being operated in a *delay-insensitive* context, as was the original circuit (*i.e.,* components and wires are assumed to have arbitrary delay). The delay-insensitive behavior of $CALL1$ is

$$\begin{aligned} CALL1 &= a? \rightarrow (c'! \parallel a'!) \rightarrow CALL1 \\ &\quad \mid b? \rightarrow (c'! \parallel b'!) \rightarrow CALL1 \end{aligned}$$

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). The circuit, $CALL1$, can be replaced by $MCALL1$ (shown in Figure 6), which is smaller and faster than $CALL1$. Clearly $MCALL1$ 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 $MCALL1 \sqsubseteq CALL1$. While the former check only guarantees that there will be no chokes if $MCALL1$ replaces $CALL1$, the latter check also assures us that $MCALL1$ can exhibit all the successful traces of $CALL1$. As a result, strong conformation insures that $MCALL1$ has *neither* the deadlock behavior illustrated in Section 4.1 *nor* the constrained output behavior illustrated in Section 4.2. Strong conformance has been used to validate a number of other optimizations in the occam synthesis system [18] as well.

4.4 Generalized Selector

An interesting phenomenon occurs when the specification for a circuit includes non-deterministic choice. Consider a *generalized selector* GS:

$$GS \quad = \quad a? \rightarrow (b! \rightarrow GS \ \mid \ c! \rightarrow GS)$$

where $\mid$ denotes *choice* (in this example, a *non-deterministic* choice). When this module receives an input on a, it makes a transition on either b or c.

Now consider the specification of an *alternating selector* [1]:

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

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

Clearly, AS is a valid replacement for GS. For example, since GS can make a non-deterministic choice, it might decide to choose strictly alternating outputs (thus, restricting its behavior to that of AS). On the other hand, it is also the case that AS *cannot* implement all of the output behaviors possible in GS.

In summary, in this example, strong conformance is too restrictive a criterion from the point of view of "safe substitution". However, if what is desired is that every trace specified by GS is possible in an implementation, then implementation AS is unacceptable; in this case, strong conformance supports the desired point of view. Thus, the appropriateness of a verification relation — conformance *vs.* strong conformance — depends precisely on the design goals being served by verification. This point is explored further in the next subsection.

4.5 A Caveat in Applying Conformance Checks

As shown in the previous examples, strong conformance can detect common errors (such as omitting a "bubble" at the input of a C-ELEMENT) which cannot be detected by conformance. However, in using the strong conformance check in practice, one must keep in mind the assumptions underlying conformance versus strong conformance.

To illustrate this point, consider the specification of a four-phase to two-phase converter with "quick return" (see Figure 7):

$$QR42\_SPEC = r4? \rightarrow ((r2! \rightarrow a2?) \parallel (a4! \rightarrow r4?)) \rightarrow a4! \rightarrow QR42\_SPEC$$

where $((a4! \to r4?) \parallel (r2! \to a2?))$ represents all possible overlapped executions of $(a4! \to r4?)$ and $(r2! \to a2?)$. This specification describes a module which converts from a *4-phase handshaking protocol* (*e.g.*, $r4? \to a4! \to r4? \to a4!$) on the left interface to a *2-phase handshaking protocol* (*e.g.*, $r2! \to a2?$) on the right interface.

Consider an implementation QR42_IMP of QR42_SPEC:

$$QR42\_IMP = r4? \to (r2! \to a2? \to a4! \to r4?) \to a4! \to QR42\_IMP$$

This implementation operates in accordance with the specification, but the concurrent behavior of QR42_SPEC has been *sequentialized.* Implementation QR42_IMP conforms to QR42_SPEC; however, QR42_IMP does *not conform strongly* to QR42_SPEC. The error-trace produced by the failed strong conformance check is `(R4 A4)`. That is, QR42_IMP is incapable of producing an `A4` immediately following an `R4`.

Depending on the application, conformance might be the appropriate verification relation, since it indicates that QR42_IMP is a safe substitution for QR42_SPEC. On the other hand, strong conformance indicates that QR42_IMP has more constrained output behavior than QR42_SPEC. In particular, QR42_IMP allows *no* concurrency between outputs $r2$ and $a4$. For certain applications, such limited behavior may be unacceptable; strong conformance successfully detects an error.

This example illustrates that the usefulness of a verification relation depends on the intended design goals. Strong conformance is not a general solution to the problem of asynchronous verification. However, for many applications, it is a simple and powerful formalism for locating errors that cannot otherwise be detected by conformance.

5 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 by Ebergen [22]. Key differences between our work and Ebergen's are noted below, and related work is also discussed.

Ebergen's trace theory is designed with different objectives: to specify computations, and synthesize circuits through *calculations* using trace-theoretic rules. This trace theory does not directly relate to circuit components; for instance, two trace structures containing the same output symbol can be *weaved.* The "weave" operator merely captures constraints on joint execution; it does not correspond to the act of connecting two circuit outputs. In contrast, Dill's $\parallel$ operator relates directly to the composition of circuit components; hence, Dill prevents the composition of two trace structures having 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*, *DI decomposition*, the *separation theorem*, and the *substitution theorem.* Together with a rich collection of equational laws on *commands* (where commands denote trace structures), Ebergen's trace theory is used to synthesize correct circuits, without having to first "generate" a circuit and then "test" it using a verifier (as has been the approach suggested here). A tool to demonstrate the power of Ebergen's trace theory, called VerDect, is now available [30]. VerDect checks for Ebergen's condition of decomposition, in effect performing a verification under the speed-independent model (delay-insensitivity is guaranteed under Ebergen's method of synthesis by performing a syntactic check on decompositions [22, 31]). Dill's and Ebergen's work 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 [22, Definition 3.1.0.0, Page 42] — as was discovered after the fact by us. A similar idea called *input liberalization* has also been proposed by Ad Peeters [32] — again discovered after the fact. However, neither Ebergen nor Peeters suggest using their definitions for validating circuit optimizations, as we do here.

An alternative methodology for translating concurrent process descriptions in a simple language into delay-insensitive circuits is described by Weber *et al.* [33]. The correctness of this compiler is shown by exhibiting a bisimulation relation [12] between the state transition system of the input description and the circuit generated from it. The authors point out that in general bisimulation is too strong an equivalence relation for use in verification. For example, although the optimization illustrated in Figure 6 is certifiable using *strong conformance*, the state transition systems of the unoptimized and the optimized circuits shown in this figure are not bisimilar. In fact, a notion of correctness identified by Dill [1] called *conformation equivalence* (defined to be true when $imp \preceq spec$ and $spec \preceq imp$), which is much weaker than the bisimulation relation, also cannot explain the relationship between the unoptimized and the optimized versions of the circuits in this figure. The fact that some correctness criteria prove to be "too strong" stems from the fact that optimizations, both at the high level as well as at the circuit level, do not usually replace equals by equals. However, bisimulation as well as conformation equivalence are correctness criteria that are useful in their own ways. Thus, we re-emphasize the generally agreed upon fact that for supporting hardware verification in practice, a *catalog* of correctness criteria is needed, and the designer should apply judgment in choosing the "right" correctness criterion for the task at hand.

The process algebra developed by Udding and Josephs holds promise to contain state explosion [34, Remark on page 2], as circuits are derived through *calculations* in their process algebra, rather than verified *post-hoc*, as with Dill's verifier. However, so long as the two points of view exist — post-hoc verification after "intelligent human design" *vs.* "correct by construction" design using *intelligent calculations* —, both approaches have an important role to play.

Finally, work in verification of asynchronous circuits appears to be proceeding along (at least) two distinct lines: (1) a class of work that uses various trace models; (2) a class of work based on process algebras. Many of the notions used in these areas seem to be conceptually similar: *e.g.*, *autofailure manifestation* [1] (which converts possible failures to actual failures) and *may/must* pre-orders (used by [17]). However there are fundamental differences between these approaches as well: *e.g.*, unidirectional wires carry information only one way, so that a component cannot refuse an input; however, a CCS/CSP rendezvous can be refused by not participating. One hopes to see unifying efforts relating these as yet unrelated efforts.

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

paper.

References

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

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

3. Jerry R. Burch. Delay models for verifying speed-dependent asynchronous circuits. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*, pages 270–274, 1992.

4. Srinivas Devadas, Kurt Keutzer, Sharad Malik, and Albert Wang. Verification of asynchronous interface circuits with bounded wire delays. In *International Conference on Computer Design (ICCAD), IEEE*, pages 188–195, November 1992.

5. Steven M. Nowick and David L. Dill. Practicality of State-Machine Verification of Speed-Independent Circuits. In *International Conference on Computer Design (ICCAD), IEEE*, pages 266–269, November 1989.

6. David E. Muller and W.S. Bartky. A Theory of Asynchronous Circuits. In *The Annals of the Computation Laboratory of Harvard University. Volume XXIX: Proceedings of the International Symposium on the Theory of Switching, Part I,* pp. 204-243, Harvard University Press, 1959.

7. John A. Brzozowski and Jo C. Ebergen. On the delay-sensitivity of gate networks. *IEEE Transactions on Computer*, 41(11):1349–1360, November 1992.

8. David L. Dill, Steven M. Nowick, and Robert F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1), July 1992.

9. Steven M. Nowick. *Personal Communication, 1992.*

10. Al Davis, Bill Coates, and Ken Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume I (Architecture and Biotechnology Computing)*, pages 409–418. IEEE Computer Society Press, 1993. *Published in the Minitrack Asynchronous and Self-Timed Circuits and Systems.*

11. A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies (Manchester, England)*, April 1993.

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

13. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

14. James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

15. Stephen H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, New York, New York, 1969.

16. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

17. Rocco DeNicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1983.

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

19. Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer Design (ICCAD), IEEE*, pages 262–265, nov 1989.

20. Martin Rem, Jan L.A. van de Snepscheut, and Jan Tijmen Udding. Trace theory and the definition of hierarchical components. In Randal E. Bryant, editor, *Proc. 1983 Caltech VLSI Conference*, pages 225–239. Computer Science Press Inc., 1983.

21. Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design.* Springer Verlag, 1985. LNCS 200.

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

23. Amir Pneuli. How vital is liveness? verifying timing properties of reactive and hybrid systems. In Rance Cleveland, editor, *Springer Verlag Lecture Notes in Computer Science, No.630, CONCUR '92*, pages 162–175. Springer Verlag, 1992.

24. Z. Har'El and Robert P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, January 1990.

25. Venkatesh Akella. Action refinement based transformation of concurrent processes into asynchronous hardware. Ph.D. research in progress.

26. 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.*

27. 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.

28. Chris Myers and Teresa H.-Y. Meng. Synthesis of timed asynchronous circuits. In *Proceedings of the International Conference on Computer Design (ICCD-92)*, pages 279–284, 1992.

29. Jerry Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie-Mellon University, August 1992. *Technical Report CMU-CS-92-179.*

30. Jo C. Ebergen and Sylvain Gingras. A verifier for network decompositions of command-based specifications. In *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume I (Architecture and Biotechnology Computing)*, pages 310–318. IEEE Computer Society Press, 1993. *Published in the Minitrack Asynchronous and Self-Timed Circuits and Systems.*

31. Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, (1):197–204, 1986.

32. Jo C. Ebergen and Ad M.G. Peeters. The modulo-n counter: Design and analysis of delay-insensitive circuits. Technical Report CS-91-25, Department of Computer Science, University of Waterloo, June 1991.

33. Sam Weber, Bard Bloom, and Geoffrey Brown. Compiling joy to silicon. In Thomas Knight and J. Savage, editors, *Advanced Research in VLSI : Proceedings of the 1992 Brown/MIT Conference.* The MIT Press, March 1992.

34. Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. Technical Report WUCS-89-54, Department of Computer Science, Washington University, St. Louis, MO, 1989.

Notation and Initialization:

- Define $\tilde{T}$ = if $(T = T_0)$ then T_1 else T_0.
- Define $next(s, x)$ to be the next state reached from state s on symbol x.
- Initialize a global set of state pairs, $visited = \phi$.
- Call *conforms-to-p* ($\overline{T_S}$,T_I, start-state($\overline{T_S}$), start-state(T_I)).

conforms-to-p-1 (T_0, T_1, st_0, st_1) =
if $(st_0, st_1) \in visited$
  then return
  else
      begin
          $visited := visited \cup \{(st_0, st_1)\}$;
          for each $T \in \{T_0, T_1\}$
              for each enabled output x of T
                    if x is enabled in $\tilde{T}$
                      then *conforms-to-p-1* $(T_0, T_1, next(st_0, x), next(st_1, x))$
                      else *ERROR (print failure trace and abort)*
                    end if
              end for
          end for
      end
end if
end *conforms-to-p-1*

conforms-to-p (T_0, T_1, st_0, st_1) =
begin
      conforms-to-p-1 (T_0, T_1, st_0, st_1);
      print("success")
end
end *conforms-to-p*

Figure 2: Algorithm for Checking Conformance

Notation and Initialization:

- Define $\tilde{T}$ = if ($T = T_0$) then T_1 else T_0.
- Define $next(s, x)$ to be the next state reached from state s on symbol x.
- Initialize a global set of state pairs, *visited* $= \phi$.
- Call *strong-conforms-to-p* ($\overline{T_S}$,T_I, start-state($\overline{T_S}$), start-state(T_I)).

strong-conforms-to-p-1 $(T_0, T_1, st_0, st_1) =$
if $(st_0, st_1) \in visited$
  then return
  else
    begin
      $visited := visited \cup \{(st_0, st_1)\}$;
      for each enabled input x of T_0 (* Strong conformance checking loop *)
        if x is not enabled in T_1
          then *ERROR (print failure trace and abort);*
        end if
      end for;
      for each $T \in \{T_0, T_1\}$
        for each enabled output x of T
          if x is enabled in $\tilde{T}$
            then *strong-conforms-to-p-1* $(T_0, T_1, next(st_0, x), next(st_1, x))$
            else *ERROR (print failure trace and abort)*
          end if
        end for
      end for
    end
end if
end *strong-conforms-to-p-1*

strong-conforms-to-p $(T_0, T_1, st_0, st_1) =$
begin
  strong-conforms-to-p-1 (T_0, T_1, st_0, st_1);
  print("success")
end
end *strong-conforms-to-p*

Figure 3: Algorithm for Checking Strong Conformance

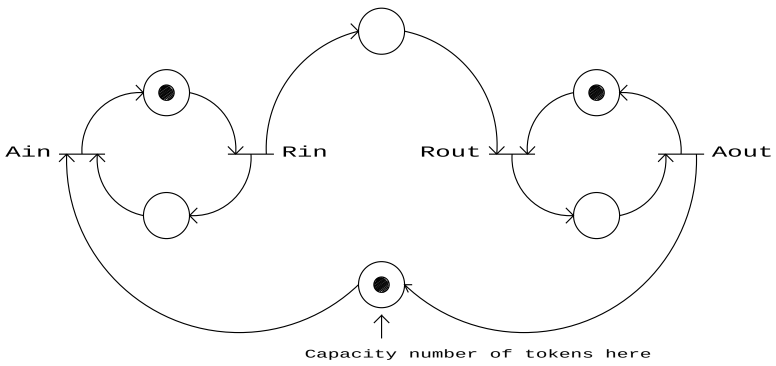

Figure 4: Petri Net Specification of a Queue

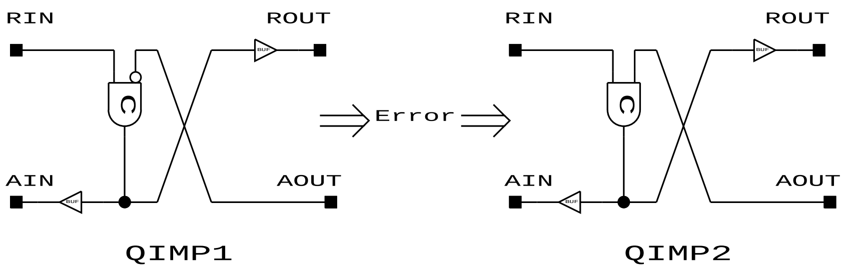

Figure 5: Two Different Queue Elements

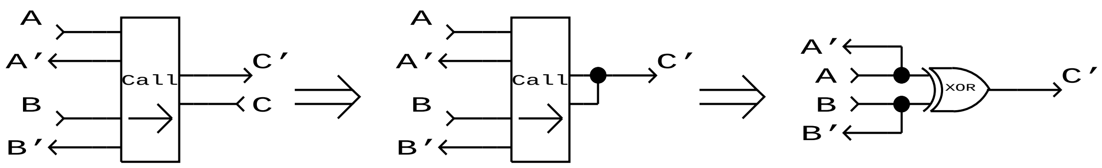

Figure 6: Call—Merge Optimization: CALL, CALL1, and MCALL1, respectively

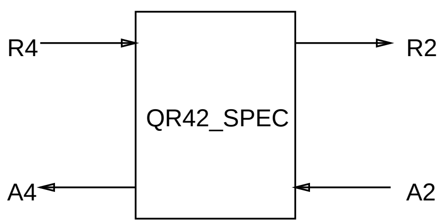

Figure 7: QR42 Converter Specification