# On process-algebraic verification of asynchronous circuits

X. Wang M. Kwiatkowska
School of Computer Science, University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
{X.Wang, M.Z.Kwiatkowska}@cs.bham.ac.uk

## Abstract

*This paper develops a theoretical basis for using process algebra and associated model checking tools to verify asynchronous circuits. We extend existing verification theory for asynchronous circuits, and integrate it into the framework of standard process algebra theory. Our theory permits analysis of safeness (i.e. choke) and progress (i.e. illegal stop, divergence and relative starvation) conditions. We show how the model can be translated into CSP, and how the satisfaction of safeness and progress requirements can be reduced to refinement checks in CSP.*

## 1 Introduction

Process algebras (e.g. CSP [12]) have been extensively used in the modelling and synthesis of asynchronous circuits, and to some extent in verification. We mention here the use of process algebra verification tools such as CWB and CADP [11, 14, 7, 10].

Our own interest in process algebraic asynchronous circuit verification stems from our adaptation of FDR2 [6] as a verification tool for the Balsa asynchronous circuit synthesis system [5].

Practically speaking, process algebras are not only good at modelling asynchronous circuits in detail; they are also good at describing the function of the circuits at a higher level of abstraction. Thus, we can program a circuit in a higher-level language (i.e. a language with synchronous blocking communication and data manipulation) and then compile down to the real circuit (low-level asynchronous non-blocking communication using signal transition). This is the approach taken by Balsa and related systems (e.g. Tangram [1]), and called *silicon compilation*.

Balsa and Tangram use as their high-level language some variant of CSP. Thus, conventional CSP theory suffices for the high-level verification (i.e. on simple abstract functional description). At the lower level (i.e. with complicated implementation details), which most asynchronous verification theories [4, 9, 15] target, an asynchronous circuit shall be regarded as a hierarchical modular system. Atomic and composite components in it are modules at different levels of the hierarchy. To tackle the state-space explosion problem, asynchronous verification theories [4, 15] usually adopt a compositional approach (i.e. in the style of assumption/guarantee). This is possible largely because, at the lower level, the communication is non-blocking, which is not the case for most process algebras.

In this paper, we develop a particular variant of compositional verification theory, called *protocol conformance verification*, for asynchronous circuits, and embed it into process algebra. Our work will follow the approach of the XDI model and Receptive Process Theory (RPT) [15, 8, 9] to retain their intuitive simplicity while at the same time addressing advanced progress conditions (*relative starvations*). One advantage of our approach is that it avoids the complexity of Dill's Complete Trace Structure construction [4].

In the rest of the paper, we specifically target CSP [12], but many of the results also apply to other process algebras. We first (in Section 2, 3, and 4) give a LTS-based general semantic model for asynchronous concurrent systems which permits the analysis of safeness (i.e. *choke*) and progress (i.e. *illegal stop*, divergence and relative starvation) conditions. We justify the model by showing how it can be translated into CSP using an infinite traces model of CSP [13] and a simple but novel idea of *a scheduler*. The construction of the scheduler (examples in Section 6) helps to avoid the need to impose process receptiveness and gives a simple proof of the semantic correspondence. Chokes are modelled as a special class of deadlock, which are distinguished from another class of good deadlock: termination. This frees CSP divergence, consequently used to model divergence and relative starvation. Relative starvation is a type of local livelocks. The challenge here is that CSP needs to distinguish good divergences from bad divergences (i.e. relative starvation). Using these new ideas, we reduce asynchronous circuit verification problems to CSP refinement checks, and prove their correctness.

Next (in Section 5), we introduce a new compositional verification theory based on the notion of protocols and conformance. Due to relative starvations, the environment and system in our theory are not symmetrical. The difficult compositionality proof is accomplished purely using set-theoretical techniques. The whole approach is illustrated by a real-world case study (Section 6), before being summarised in Section 7.

## 2 Asynchronous concurrent systems

Our theory will be based on a general LTS model of asynchronous concurrent systems, though terms like gates and circuits are used to give the frame of reference. We will later show that our theory coincides with and extends [4, 8, 9, 15]. Therefore, it supports asynchronous circuit verifications.

A *IOTS* (*Input/Output Transition System*) is a tuple $(I, O, S, T, s^0)$, consisting of

- a finite set of input events, $I$, called the input alphabet.
- a finite set of output events, $O$, called the output alphabet ($I$ and $O$ are disjoint and their union is $A$).
- a finite set of states, $S \cup \{Choke\}$.
- a transition relation $T : S \times (A \cup \{\tau\}) \leftrightarrow (S \cup \{Choke\})$. $\tau$ is the invisible event. Given a state and an event, $T$ nondeterministically produces a new state.
- $s^0 \in S \cup \{Choke\}$ is the initial state.

In the rest of the paper, $(\!|\,|\!)$ is the image operation on relations. Let $e$ denote a non-$\tau$ event, $a$ denote a $\tau$ or non-$\tau$ event, and $\Delta$ denote a subset of $A$. $t$ and $u$ are finite traces, i.e. finite sequences of non-$\tau$ events (including the empty sequence, $\epsilon$)[1], i.e. $t, u \in A^*$. $\bar{t}$ and $\bar{u}$ are the infinite variants, i.e. $\bar{t}, \bar{u} \in A^\omega$. Juxtaposition is used for sequence concatenation, e.g. $tu$.

**Definition 2.1.** Given a IOTS and some of its states, e.g. $s$ and $s'$:

- $s \xrightarrow{a} s'$ iff state $s$ has an outgoing transition labeled by $a$ to $s'$.
- $s \xrightarrow{a}$ iff there exists a state $s'$ in the IOTS such that $s \xrightarrow{a} s'$.
- for a state $s$, the set of enabled inputs is $ETS^I = \{e : I \mid s \xrightarrow{e}\}$, the set of enabled outputs is $ETS^O = \{e : O \mid s \xrightarrow{e}\}$, and the set of enabled (visible) events is $ETS = ETS^I \cup ETS^O$.
- $s \rightarrow$ iff there exists a state $s'$ and event $a$ in the IOTS such that $s \xrightarrow{a} s'$. When $\neg(s \rightarrow)$, we say $s$ is a *sink state* (i.e. a deadlock state).
- $\xrightarrow{a} s'$ iff there exists a state $s$ in the IOTS such that $s \xrightarrow{a} s'$, and we say $s'$ is *caused* by $a$.
- a state $s$ is *stable* iff $\neg(s \xrightarrow{\tau})$.
- a state $s$ is *divergent* iff there is a zero or nonzero step $\tau$-path going from $s$ to a state $s'$ such that $s'$ is in a $\tau$-loop.
- a state $s$ is *infinite output enabled (ioe)* iff there is a zero or nonzero step output-path (i.e. a path consisting of only output transitions) going from $s$ to a state $s'$ such that $s'$ is in an output-loop (i.e. a loop consisting of only output transitions).
- $s \overset{\epsilon}{\Longrightarrow} s'$ iff there is a zero or nonzero step $\tau$-path going from $s$ to $s'$.
- $s \overset{e}{\Longrightarrow} s'$ iff there are $s_0$ and $s_1$ such that $s_0 \xrightarrow{e} s_1$, $s \overset{\epsilon}{\Longrightarrow} s_0$ and $s_1 \overset{\epsilon}{\Longrightarrow} s'$.
- $s \overset{t}{\Longrightarrow} s'$, when $t = et_0$, iff there is $s_0$ such that $s \overset{e}{\Longrightarrow} s_0$ and $s_0 \overset{t_0}{\Longrightarrow} s'$.
- $s \overset{\bar{t}}{\Longrightarrow}$ iff there is an infinite sequence of states $s_0 s_1 s_2 ...$ such that $s \overset{e_0}{\Longrightarrow} s_0$, $s_0 \overset{e_1}{\Longrightarrow} s_1$, $s_1 \overset{e_2}{\Longrightarrow} s_2$ ..., where $\bar{t} = e_0 e_1 e_2 ....$
- $t \restriction_A$ projects a trace $t$ onto an alphabet $A$ by removing from $t$ all the events not in $A$ while keeping the order. It gives us another trace in $A^*$. $\restriction_A$ can be lifted to operate on infinite traces or sets of traces.

**Definition 2.2.** A IOTS is *divergence-free* iff there is no divergent state reachable in it. A IOTS is *choke-free* iff *Choke* is not reachable. A IOTS is *deadlock-free* iff there is no deadlock state reachable. A IOTS is *$\tau$-free* iff there is no reachable state that has an outgoing $\tau$ transition.

A *LAG* (*Logical Abstract Gate*) is a *$\tau$-free and choke-free IOTS*. This essentially means that we can use any formalism (e.g. Petri net [2, 17] or process algebra [12]) to specify it as long as the formalism can be given a LTS semantics. From LAGs, a *LAC* (*Logical Abstract Circuit*) can be built using hiding and parallel operators:

$$LAC ::= LAG \mid LAC \parallel LAC' \mid LAC \setminus \Delta_O$$

We view a LAC as an asynchronous circuit in which the coupling between the components is defined via their *alphabets*. To avoid two components outputting on the same wire, output alphabets should be disjoint. Parallel composition can only compose *non-interfering* LACs, i.e. $O(LAC) \cap$

[1] Sometimes, $a$ and $e$ are also used as singleton sequence and singleton trace.

$O(LAC') = \{\}$. On the other hand, components are allowed to share inputs (i.e. broadcast communication in an isochronical fork). If an event is both an input and output (i.e. for different components), it will be an output for the circuit. Hiding can only hide events in the output alphabet of LAC, i.e. $\Delta \subseteq O(LAC)$. If the input alphabet is empty, the LAC will be a *complete* or *closed* circuit.

**Definition 2.3.** Given non-interfering $IOTS_1$ and $IOTS_2$, $IOTS_1 \parallel IOTS_2$ gives another IOTS, $(I, O, S \cup \{Choke\}, T \cup K, s^0)$, where $O = O_1 \cup O_2$, $I = (I_1 \cup I_2) \setminus O$, $A = A_1 \cup A_2$, $S = S_1 \times S_2$,

$$s^0 = \begin{cases} Choke & \text{either } s_1^0 \text{ or } s_2^0 \text{ is } Choke \\ (s_1^0, s_2^0) & \text{otherwise} \end{cases}$$

$T : S \times A \cup \{\tau\} \leftrightarrow S$ is the least relation satisfying the rules (note that $s$, $s'$ and $s_i$ below range over non-choke states):

$$\frac{s_1 \xrightarrow{a} s_1' \quad a \notin A_2}{(s_1, s_2) \xrightarrow{a} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{a} s_2' \quad a \notin A_1}{(s_1, s_2) \xrightarrow{a} (s_1, s_2')} \qquad \frac{s_1 \xrightarrow{e} s_1' \quad s_2 \xrightarrow{e} s_2'}{(s_1, s_2) \xrightarrow{e} (s_1' s_2')}$$

and $K : S \times O \cup \{\tau\} \leftrightarrow \{Choke\}$ is the least relation satisfying the rules:

$$\frac{s_1 \xrightarrow{e} s_1' \quad \neg(s_2 \xrightarrow{e}) \quad e \in O_1 \cap I_2}{(s_1, s_2) \xrightarrow{e} Choke} \qquad \frac{s_1 \xrightarrow{a} Choke}{(s_1, s_2) \xrightarrow{a} Choke}$$

$$\frac{s_2 \xrightarrow{e} s_2' \quad \neg(s_1 \xrightarrow{e}) \quad e \in O_2 \cap I_1}{(s_1, s_2) \xrightarrow{e} Choke} \qquad \frac{s_2 \xrightarrow{a} Choke}{(s_1, s_2) \xrightarrow{a} Choke}$$

Except for the ones related to *Choke* (the last four rules), the transition rules are consistent with those for the alphabetised parallel operator in CSP. *Choke* is a special state: choke states are created by a mismatch of input and output, and local choke implies global choke.

The hiding and the renaming operators are consistent with those of CSP.

Assume an infinite set of LAC variables (typed by their alphabets) $X$ ranged over by $x$. A *Generalized Context GC* is defined as:

$$GC ::= LAG \mid x \mid GC \parallel GC' \mid GC \setminus \Delta$$

where any variable $x$ in $GC$ must have a unique occurrence.

Often we use $\vec{x}$ to represent the vector of variables in $GC$, and, to make them explicit, $GC$ can be equivalently written as $GC[\vec{x}]$. If there is no LAG occurring in $GC$ (i.e. $GC$ is made up of only variables and operators), we say it is a *super-combinator*, written as $SC[\vec{x}]$. If $GC$ has only one variable, we say it is a *context* and the variable is its *hole*, written as $C[\cdot]$. Given $GC[\vec{x}]$ and a vector $\vec{LAC}$ of *compatible* LACs, substituting $\vec{LAC}$ for $\vec{x}$ in $GC$ gives us a new LAC, written as $GC[\vec{LAC}]$. It is easy to see that any LAC can be written in the form of $SC[\vec{LAG}]$.

**Definition 2.4.** $\vec{LAC}$ is *compatible* with $GC[\vec{x}]$ iff each $LAC$ in $\vec{LAC}$ has the same alphabet as that of the corresponding $x$ in $\vec{x}$.

**Theorem 2.5 ([4]).** *Given $SC[\vec{LAG}]$, it has a normal form, $(\|[\vec{LAG'}]) \setminus \Delta$, where $\vec{LAG}$ and $\vec{LAG'}$ are of the same dimension and $\|$ is a super-combinator consisting of only parallel operators, such that*

1. *the corresponding members of $\vec{LAG}$ and $\vec{LAG'}$ are renaming-isomorphic.*
2. *$SC[\vec{LAG}]$ and $(\|[\vec{LAG'}]) \setminus \Delta$ are isomorphic.*

Note that $IOTS$ and $IOTS'$ are renaming-isomorphic iff there is a 1-to-1 renaming on one of them and the resulting IOTS is isomorphic to the other.

## 3 The CSP model

In this paper we assume familiarity with CSP [12]. Stable failures and failure/divergences are the major semantic models used in this section. They are both finite trace models. However, it is worth mentioning that there is a newly developed infinite trace CSP model [13], the $\mathcal{SBD}$ model. In this model, the denotation of each process consists of three components $(F, I, D)$. $F$ is the failures set as in the stable failures model, $I$ is the infinite traces set, and $D$ is the divergences set. $\mathcal{SBD}$ gives, for the first time, a congruence that preserves all the divergences in CSP processes. This result will become crucial in Section 4 when we are dealing with the translation of circuits with advanced progress requirements, i.e. freedom of relative starvations.

We now show how the model of asynchronous circuits introduced in Section 2 can be translated into CSP using a simple construction of a scheduler. A scheduler helps to model the *Choke* state as a special CSP deadlock, avoiding the receptiveness requirement on processes.

The basic idea is that *all events* (including $\tau$) are split into *two* events, e.g. $e$ into $e^i$ and $e^o$, and $\tau$ into $\tau^i$ and $\tau^o$ [2]. $e^o$ (or $\tau^o$) is used by the sending side (one LAG) of $e$ while $e^i$ and $\tau^i$ by the receiving side (one or many LAGs).

For instance, given $LAG = (I, O, S, T, s^0)$, it can be translated into CSP via the mapping $CSP(LAG) \mathrel{\widehat{=}} P(s^0)$[3], where

$$P(s) = \Box(e, s') : \{(e, s') \mid s \xrightarrow{e} s'\} \\ \bullet \text{ if } e \in I \text{ then } e^i \to P(s') \text{ else } e^o \to P(s')$$

Then, a complete circuit[4], $LAC$, is translated into $\Omega(LAC) = CSP(LAC) \parallel scheduler(O \cup \{\tau\})$, a synchronous parallel composition (i.e. synchronization on all events) of a scheduler and $CSP(LAC)$, where:

[2]Note that, unlike $\tau$, $\tau^i$ and $\tau^o$ should be treated as normal events.

[3]We assume here that recursion will not introduce extra $\tau$ events.

[4]Note that only complete LACs need to be translated since our verification theory in Section 5 will reduce any verification problem to checks on some complete LAC.

$$
\begin{aligned}
&\mathit{scheduler}(X) \\
&= \ \Box\, a : X \bullet a^o \rightarrow a^i \rightarrow \mathit{scheduler}(X)
\end{aligned}
$$

The scheduler synchronizes with all the signal transitions in *LAC* to control their occurrences. It first checks the to-be-sent signal transitions available, selects one, e.g. $a^o$, and then forces it onto the receiving side using $a^i$. In case of deadlocks when forcing $a^i$, this implies that the *Choke* state is reached. Otherwise, i.e. deadlock when selecting $a^o$, this implies the termination of the circuit execution.

The translation of *LAC* is given by:

$$CSP(LAC_1 \parallel LAC_2) = CSP(LAC_1) \,[\![ I_1^i \cap I_2^i ]\!]\, CSP(LAC_2)$$

$$CSP(LAC \setminus \Delta) = (CSP(LAC) \,[\![ \Delta^o \cup \Delta^i ]\!]\, \mathit{scheduler}(\Delta)) \\ [e : \Delta \bullet e^i \leftarrow \tau^i, e^o \leftarrow \tau^o]$$

$[\![\Delta]\!]$ is the binary interface parallel operator [12]. Its two arguments synchronize on, and only on, events in $\Delta$. Note that, before hiding (i.e. $\setminus\ \Delta$), we need to supply a local scheduler on to-be-hidden events and hiding is defined in terms of renaming to $\tau^i$ and $\tau^o$.

Splitting events and introducing a scheduler may seem unnatural. But, it is actually not a problem if we put it into the perspective of real circuit applications, where an event is usually associated with a wire. A wire connects two components, each component shall have a different local name for the wire (one as input and one as output). When specifying a protocol, it is good practice to use the local names rather than the global name, since it implies the independence from the wiring configuration and facilitates reuse. Therefore, when composing a set of protocols, we still need some mechanism to encode the wiring configuration and link them together [5]. A scheduler is very natural to play the role. Section 6 gives a full example using schedulers for choke-freedom verification.

**Definition 3.1.** Given a complete *LAC* and $\Omega(LAC)$, a state $s$ in $\Omega(LAC)$ is *transient* iff there exists an event $a \in O^o \cup \{\tau^o\}$ such that $( \xrightarrow{a} s)$.

**Proposition 3.2.** *Given a complete LAC and $\Omega(LAC)$, all transient states have exactly one incoming transition and one or zero outgoing transitions.*

*Proof.* Induction on the structure of *LAC*. □

**Proposition 3.3.** *Given a complete LAC and $\Omega(LAC)$, the transitions labeled by $a^o$, where $a \in O \cup \{\tau\}$, are from non-transient states to transient states; the transitions labeled by $a^i$ are from transient states to non-transient states.*

*Proof.* Induction on the structure of *LAC*. □

[5]For instance, it can be implemented using linked parallel operator of CSP.

**Proposition 3.4.** *Given a complete LAC and $\Omega(LAC)$, there exists a bijection f from the set of non-choke states of LAC to the set of non-transient states of $\Omega(LAC)$, and a bijection k from the set of transitions of LAC to the set of transient states of $\Omega(LAC)$, such that:*

- *for any transition $s \xrightarrow{a} s'$ of LAC, where $s' \neq$ Choke, there is a pair of transitions $f(s) \xrightarrow{a^o} k(s,a,s')$ and $k(s,a,s') \xrightarrow{a^i} f(s')$ in $\Omega(LAC)$, and*
- *for any transition $s \xrightarrow{a}$ Choke of LAC, there is a transition $f(s) \xrightarrow{a^o} k(s,a,s')$ and $k(s,a,s')$ is a sink state in $\Omega(LAC)$.*

*Proof.* Note that each state in $\Omega(SC[\vec{LAG}])$ is labeled by $\vec{s}$. □

**Theorem 3.5.** *Given LAC and $\Omega(LAC)$, LAC is choke-free iff $\Omega(LAC)$ is free of deadlock transient states, or iff $\Omega(LAC)$ has no deadlock trace ending with some $a^o$, i.e. $goodsched(O) \sqsubseteq_F \Omega(LAC)$ in the stable-failures model, and*

$$
\begin{aligned}
goodsched(O) = \ & Stop \sqcap \\
& \Box\, a : O \cup \{\tau\} \bullet (a^o \rightarrow a^i \rightarrow goodsched(O))
\end{aligned}
$$

*Proof.* Use Proposition 3.4. □

**Theorem 3.6.** *Given LAC and $\Omega(LAC)$, LAC is divergence-free iff $\Omega(LAC) \setminus \{\tau^i, \tau^o\}$ is divergence-free in the failure/divergences model.*

*Proof.* Use Proposition 3.4. □

# 4 Safeness and progress conditions

Compared to synchronous circuits, the design of asynchronous circuits faces a major challenge: *glitches* [3]. A glitch happens when there exists an assignment of delays that can cause an unwanted signal transition to occur in the circuit. The transmission interference is a type of glitch.

In our framework, each component in a complete circuit is assigned a 'legal' behaviour, which specifies both its possible output behaviours and acceptable input behaviours. For instance, for a buffer component, an input will enable its further output. If the input is always enabled, consecutive inputs will cause transmission interference by enabling multiple copies of output events. This can be avoided by defining the legal behaviour to enable the input only after the current output finishes (i.e. output event occurs). Therefore, when an undesirable signal transition is produced (i.e. consecutive inputs), it will be detected due to the choke on the receiving component (i.e. the buffer). Choke-freedom is *safeness*, one of the main correctness requirements of asynchronous circuits.

However, the above framework is inadequate for capturing another two important but less well-understood types of circuit design errors: *illegal stops* and *relative starvation*. An illegal stop occurs when there exists an execution scenario in the (complete) circuit such that no component in the circuit will produce any transitions, i.e. a stop, while some components are still waiting for the incoming signals. A relative starvation on a component occurs when there exists an execution scenario such that the component is waiting for the incoming signals while the rest of the circuit (i.e. its environment) lapsed into an infinite sequence of internal activities (i.e. a livelock). Unlike choke and illegal stop, which are global phenomena of a complete circuit, relative starvation is a local phenomenon. A complete circuit is free of relative starvations iff there is no relative starvation on any of its components. A circuit free of illegal stops and relative starvations satisfies what is called *progress conditions*. In other words, safeness makes sure no unexpected input will ever occur, while progress makes sure that expected input will eventually occur. The LAG and LAC models are inadequate in this respect because they only have a global view of livelocks (i.e. divergence) and do not differentiate illegal livelocks/stops from the legal ones.

Similarly to the approach adopted in [15], we can extend a LAG by labeling its states to indicate when it has a progress requirement on the environment. Thus, a stop or relative livelock will be legal iff it happens when none of the components in the circuit have any progress requirement on the environment.

Therefore, we arrive at the following generalized LAG and LAC models. A *GLAG* (*Generalised Logical Abstract Gate*) is:

$$GLAG = (LAG, Q, QH)$$

where $Q$ ($\subseteq S$) labels the set of *quiescent* states of *LAG*, and $QH$ ($\subseteq Q$) labels the set of *quiescent hunger* states. A state of *LAG* is quiescent iff it *may* stop to produce output. A state is a hunger state iff it *may* need progress requirement on the environment. A state is a quiescent hunger state iff it *may* be quiescent and in a hunger state simultaneously.

Labeling states to indicate quiescence seems redundant here, since in our model stops have been intuitively captured by the emptiness of $ETS^O$ (i.e. the set of enabled output events); they are 'must' stops. Our reason for introducing quiescence is that we need to introduce a special operation on GLAGs, *DT*, which will render the interpretation based on $ETS^O$ invalid and introduce 'may' stops.

Well-formedness conditions need to be extended as well by requiring:

- All 'must' stop states in the LAG must be quiescent as well.

A GLAG defines a finite-state LTS. A GLAG with a nondeterministic LTS can be transformed to a GLAG with a deterministic LTS using a determinization operation *DT* similar to the subset construction. Such deterministic GLAGs are called *transparent*.

**Definition 4.1.** Given a *GLAG* or *LAG*, it is *transparent* iff $T$ is a partial function.

**Definition 4.2.** Given $GLAG' = DT(GLAG)$, we have

- $I' = I, O' = O, S' = \mathbb{P}(S), s'^0 = \{s^0\}$
- $Q' = \{s' \mid \exists s \in s' \bullet s \in Q\}$ and $QH' = \{s' \mid \exists s \in s' \bullet s \in QH\}$
- $T' = \{(s'_0, e) \mapsto s'_1 \mid s'_1 = R_e(\!| s'_0 |\!) \wedge e \in ets(s'_0)\}$

where $R_e = \{s_0 \mapsto s_1 \mid (s_0, e) \mapsto s_1 \in T\}$ and $ets(s') = \{o : O \mid \exists s \in s' \bullet o \in ETS(s)\} \cup \{i : I \mid \forall s \in s' \bullet i \in ETS(s)\}$.

The *DT* operation may introduce extra disabling actions, i.e. output disabling output. *DT* can be lifted to operate on a vector of GLAGs.

**Proposition 4.3.** *DT is closed on the well-formed GLAGs.*

Other propositions about the *DT* operation can be found in Section 5.

Based on GLAGs, we can define *GLACs* (*Generalised Logical Abstract Circuits*) as follows:

$$GLAC ::= GLAG \mid GLAC \parallel GLAC' \mid GLAC \setminus \Delta \,.$$

The remaining definitions including super-combinators and transition rules of LTSs induced by GLACs are essentially the same as those for LACs.

A GLAC is a set of properly coupled GLAGs. It is in a quiescent state iff all its GLACs are in quiescent states. It is in a quiescent hunger state iff all its GLAGs are in quiescent states and at least one of them is also in a quiescent hunger state.

**Definition 4.4.** A state in a complete GLAC is an *illegal stop* iff it is a quiescent hunger state. A complete GLAC is *free of illegal stops* iff no illegal stop is reachable in GLAC.

Given a complete circuit $SC[\vec{GLAG}]$, its syntactic structure also implies a particular point of view, which becomes obvious if we look at its normal form, $(\|[\vec{GLAG'}]) \setminus \Delta$. $A' \setminus \Delta$ defines the point of view of $SC[\vec{GLAG}]$ ($A'$ is the alphabet of $\|[\vec{GLAG'}]$). By changing the $\Delta$, we can change the point of view. For example, given the complete circuit $\|[\vec{GLAG'}]$, if we want to take the point of view of $GLAG'_i$ in $\vec{GLAG'}$, then the new system should be $\|[\vec{GLAG'}] \setminus (A' \setminus A'_i)$, where $A'_i$ is the alphabet of $GLAG'_i$.

**Definition 4.5.** Given a complete circuit $|||[\vec{GLAG}]$ and a variable $x$ occurring in its super-combinator (i.e. $|||[\vec{x}]$), $|||[\vec{GLAG}]$ is *free of relative starvation on x* iff $|||[\vec{GLAG'}] \setminus (A \setminus A_x)$ cannot reach a divergent state, where $GLAG_x$ (i.e. the one corresponding to $x$ in $\vec{GLAG}$) is in a quiescent hunger state, or equivalently, cannot have a divergence trace that can reach a quiescent hunger state of $GLAG_x$.

**Definition 4.6.** A complete circuit $SC[\vec{GLAG}]$ is *free of relative starvations* iff there is no relative starvation on any $x$ occuring in $SC$.

To model quiescence and quiescent hunger states in CSP, we introduce a new pair of input/output events: $\surd^i$ and $\surd^o$. Like $\tau^i$ and $\tau^o$, they are just normal events (not $\surd$ in CSP). Whenever a GLAG is in a quiescent state, it indicates quiescence by enabling $\surd^o$ events. Whenever a GLAG is not in a quiescent hunger state, it indicates its willingness to accept quiescence by enabling $\surd^i$ events. Unlike other output events of GLACs, but like $\surd$ of CSP, $\surd^o$ will be in the alphabet of all GLAGs contained in a GLAC and be synchronized upon. Based on these events, we can adapt the translation functions, $CSP$ and $\Omega$, to complete GLACs as follows (only the modified rules are listed):

$$\Omega(GLAC) = CSP(GLAC) \parallel scheduler(O \cup \{\tau, \surd\})$$

$$\begin{aligned} &CSP(GLAC_1 \parallel GLAC_2) = CSP(GLAC_1) \\ &\quad [\![ (I_1^i \cap I_2^i) \cup \{\surd^i, \surd^o\} ]\!]\ CSP(GLAC_2) \end{aligned}$$

and $CSP(GLAG) \mathrel{\widehat{=}} P(s^0)$, where

$$\begin{aligned} P(s) = \;& \Box(e,s') : \{(e,s') \mid s \xrightarrow{e} s'\} \\ &\quad \bullet \text{ if } e \in I \text{ then } e^i \to P(s') \text{ else } e^o \to P(s') \\ &\ \Box \text{ if } s \notin Q \text{ then } Stop \text{ else } \surd^o \to \\ &\qquad (\text{if } s \notin QH \text{ then } \surd^i \to Stop \text{ else } Stop) \end{aligned}$$

$P(s)$ is changed by adding the second operand of the binary external choice operator to handle quiescence and hunger states.

All the above changes will not affect Theorem 3.6 concerning divergences. However, now we have two types of sink transient states in $\Omega(GLAC)$, one caused by $\surd^o$ and the other by $a^o$ where $a \neq \surd$. They can be checked collectively and we have:

**Theorem 4.7.** *Given GLAC and $\Omega(GLAC)$, GLAC is free of chokes and illegal stops iff $\Omega(GLAC)$ is free of sink transient states caused by $a^o$, or iff $\Omega(GLAC)$ has no deadlock trace ending with $a^o$, i.e. $goodsched(O) \sqsubseteq_F \Omega(GLAC)$ in the stable-failures model, and*

$$\begin{aligned} goodsched(O) = \;& Stop \sqcap \\ & \Box\, a : O \cup \{\tau, \surd\} \bullet (a^o \to a^i \to goodsched(O)) \end{aligned}$$

*Proof.* Use Proposition 3.4 which can be largely carried over to the new $\Omega$, except for the part related to $\surd$. □

**Theorem 4.8.** *Given a complete circuit $|||[\vec{GLAG}]$ (free of chokes and illegal stops) and a variable $x$ occurring in its super-combinator (i.e. $|||[\vec{x}]$), $|||[\vec{GLAG}]$ is free of relative starvations on $x$ iff $K(DT(GLAG_x)) \sqsubseteq_{SBD} \Omega(|||[\vec{GLAG}] \setminus (A \setminus A_x)) \setminus \{\tau^i, \tau^o, \surd^o, \surd^i\}$, where $K(GLAG) \mathrel{\widehat{=}} k(s^0)$, and*

$$\begin{aligned} k(s) = \;& \Box(e,s') : \{(e,s') \mid s \xrightarrow{e} s'\} \bullet\ e^o \to e^i \to k(s') \\ &\quad \sqcap \text{ if } s \in QH \text{ then } Stop \text{ else } (\mu\, l.(l) \sqcap Stop) \end{aligned}$$

*Proof.* Recall that, in the $\mathcal{SBD}$ model, a process is denoted by $(F, I, D)$ and refinement is component-wise supersethood. □

## 5 Protocol conformance verification

An asynchronous circuit is regarded as a hierarchical modular system. Atomic components (i.e. GLAGs) and composite components (i.e. GLACs) in it are modules at different levels of the hierarchy. Each of them is associated with a *protocol*.

A protocol can be defined as an arbitrator sitting at the boundary of the system (i.e. GLACs) and environment (see below) and observing the interaction between the system and environment.

**Definition 5.1.** An *environment* is a complete context, i.e. a context that, once filled, becomes a complete circuit.

The arbitrator must be *transparent* to avoid arbitrariness in judgement, and it must be able to distinguish the actions of the system from the actions of the environment. Formally, a protocol is given by:

$$PROT \mathrel{\widehat{=}} ((I_E, I_S), LAG^D, (Q_E, Q_S))$$

where $LAG^D$ is a transparent LAG with $O = \{\}$ (i.e. an observer), $I_E \cup I_S = I$ and $Q_E \cup Q_S = S$ (i.e. in any state of $PROT$ at least one side can quiesce).

Well-formedness conditions on LAGs need to be extended as follows:

- All states with $ETS \cap I_E = \{\}$ in the LAG must be in $Q_E$ as well.
- All states with $ETS \cap I_S = \{\}$ in the LAG must be in $Q_S$ as well.

The relationship between protocols and components (i.e. GLACs) should be understood in the following way:

- An assumption/guarantee approach to specifying the behaviour of components: a circuit component produces correct outputs so long as the environment provides legal inputs. A 'better than' relationship, i.e. less assumption and more guarantee.

- When modelling a component in a system, we do not need to model the full behaviour of the component; we only need to model the particular 'use' of the component in the system, that is, the protocol of the component. This is an under-approximation; we may get false negatives.

Given a protocol, we can derive a pair of *instrumentation systems*, one acting as the *environment* to instrument systems for their conformance to the protocol, and the other acting as the *system* to instrument environments for their conformance.

**Definition 5.2.** Given $PROT = ((I_E, I_S), (I, \{\}, S \cup \{Choke\}, T, s^0), (Q_E, Q_S))$, define $SYS(PROT) = ((I_E, I_S, S \cup \{Choke\}, T, s^0), Q_S, Q_S \setminus Q_E)$ and $ENV(PROT) = [\cdot] \parallel TM(PROT)$, where $TM(PROT) = TL(((I_S, I_E \cup \{d\}, S \cup \{Choke\}, T, S^0), Q_E, Q_E \setminus Q_S))$, and $TL(GLAG)$ give us another GLAG whose LTS is exactly like that of $GLAG$ but with all $Q$ states added a $d$-labeled self-loop.

Relative to four undesirable observables, i.e. divergences, chokes, illegal stops and relative starvations (DCIRs), protocol conformance is defined using the correct instrumentation of a component by $ENV$ and $SYS$:

**Definition 5.3.** Given $PROT$, *GLAC conforms to PROT* iff $GLAC$ is of the same alphabet as $SYS(PROT)$ and $ENV(PROT)[GLAC]$ is free of DCIRs. Dually, an environment $C[\cdot]$ *conforms to PROT* iff its hole is of the same alphabet as $SYS(PROT)$ and $C[SYS(PROT)]$ is free of DCIRs.

Based on protocol and conformance, a hierarchical system, represented as a n-level tree with each node associated with a protocol, can be verified in the following way: each leaf component (i.e. a GLAG at level 1) is assumed or verified to conform to the associated protocol. Then for each node at level 2, which is the parent of a number leaf nodes, use its protocol to derive $ENV$ and instrument the parallel composition of the sons' protocol. If correct, it implies that the component at level 2 conforms to its protocol. This can go on until it reaches the root level (i.e. level n). Therefore, it breaks a large global verification problem (concerning the root node protocol and the whole hierarchy) into a set of small local problems (each concerning only one parent node protocol and its son node protocols). The correctness of the verification method is ensured by Theorem 5.12 at the end of this section.

But before we can work on its proof, we first need define a refinement (i.e. better than) relation ($\preceq_S$) between components. The idea is that Theorem 5.12 can be derived from the fact that $ENV$ and $SYS$ are respectively the worst environment and system conforming to a protocol (i.e. Proposition 5.11).

**Definition 5.4.** Given $GLAC$ and $GLAC'$ of the same alphabet, $GLAC \preceq_S GLAC'$ (i.e. *GLAC refines*, or *is better than*, $GLAC'$) iff, in any compatible environment $C[\cdot]$, $C[GLAC']$ is free of DCIRs implies that $C[GLAC]$ is free of DCIRs. When $GLAC \preceq_S GLAC'$ and $GLAC' \preceq_S GLAC$, we say $GLAC$ and $GLAC'$ are $\preceq_S$-equivalent.

**Definition 5.5.** Given environments $C[\cdot]$ and $C'[\cdot]$ with holes of the same alphabet, $C[\cdot] \preceq_E C'[\cdot]$ (i.e. *C refines*, or *is better than*, $C'$) iff, for any compatible $GLAC$, $C'[GLAC]$ is free of DCIRs implies that $C[GLAC]$ is free of DCIRs. When $C[\cdot] \preceq_E C'[\cdot]$ and $C'[\cdot] \preceq_E C[\cdot]$, we say $C$ and $C'$ are $\preceq_E$-equivalent.

**Proposition 5.6.** *$\preceq_S$ and $\preceq_E$ are preorders.*

*Proof.* Follows from Definition 5.4 and 5.5. □

An important property is that replacing a component by a better one in any context will only improve the system, which is obtained from the following.

**Proposition 5.7.** *$\preceq_S$ is a pre-congruence.*

*Proof.* Suppose $GLAC \preceq_S GLAC'$. Given any compatible context $C[\cdot]$, it is easy to see $C[GLAC] \preceq_S C[GLAC']$ since for any compatible environment $E[\cdot]$ of $C[GLAC]$ and $C[GLAC']$, $E[C[\cdot]]$ is a compatible environment of $GLAC$ and $GLAC'$ and so $E[C[GLAC]] \preceq_S E[C[GLAC']]$. □

Our proof will be set theoretical and use the following partitioning of the (finite and infinite) trace space.

**Definition 5.8.** Given a GLAC in a restricted form (i.e. without hiding), denoted $NC \mathrel{\widehat{=}} \|[\vec{GLAG}]$, assume it has an alphabet $A = I \cup O$, an initial state $s^0$, and quiescent and quiescent-hunger state sets $Q$ and $QH$. Then, $A^\infty = A^\omega \cup A^*$ can be partitioned by:

$$
\begin{aligned}
&UIA = \{t' \mid \exists\, t, s, i \bullet s^0 \stackrel{t}{\Longrightarrow} s \wedge t' = tiu \wedge \neg\,(s \stackrel{i}{\rightarrow}\ )\}\\
&UOA = \{t' \mid \exists\, t, o \bullet \forall\, s : s^0 \stackrel{t}{\Longrightarrow} s \bullet\\
&\qquad\qquad t' = tou \wedge \neg\,(s \stackrel{o}{\rightarrow}\ )\} \setminus UIA\\
&TRC = A^\infty \setminus (UIA \cup UOA)
\end{aligned}
$$

where $u$ is any event sequence including $\epsilon$ and infinite sequences. $UIA$ (Unexpected Input Arrival) is the extension closure of the set of choke traces. $UOA$ (Unexpected Output Arrival) is the set of impossible traces (due to impossible output) that do not belong to $UIA$. $TRC$ is the set of safe traces (i.e. not violating safeness).

$TRC$ can be further partitioned into:

$$
\begin{aligned}
&EIN = \{t \mid \exists\, s \bullet s^0 \stackrel{t}{\Longrightarrow} s \wedge s \in QH\} \setminus UIA\\
&EON = \{t \mid \forall\, s : s^0 \stackrel{t}{\Longrightarrow} s \bullet s \notin Q\} \setminus UIA\\
&PT = \{t \mid \exists\, s \bullet s^0 \stackrel{t}{\Longrightarrow} s \wedge s \in Q \setminus QH\} \setminus (UIA \cup EIN)\\
&PI = \{\bar{t} \mid \forall\, t : t < \bar{t} \bullet \exists\, s \bullet s^0 \stackrel{t}{\Longrightarrow} s\} \setminus UIA
\end{aligned}
$$

*EIN* (Expected Input Never arrive) is the set of safe hunger traces (i.e. environment is not allowed to stop after these traces). *EON* (Expected Output Never arrive) is the set of safe non-stopping traces (i.e. system is not allowed to stop after these traces). *PT* is the set of safe stoppable traces (i.e. both system and environment are allowed to stop). *PI* is the set of safe infinite traces.

**Proposition 5.9.** *Given $GLAG' = DT(GLAG)$, $GLAG'$ and $GLAG$ are $\preceq_S$-equivalent.*

*Proof.* $DT$ operation is essentially a subset construction procedure. It preserves all the traces sets: *UIA*, *UOA*, *EIN*, *EON*, *PT*, and *PI*. □

**Proposition 5.10.** *Given PROT, $SYS(PROT)$, $TM(PROT)$*[6]*, and a relation on trace sets defined as: $S \trianglelefteq S'$ iff $S \subseteq S' \wedge S = S' \upharpoonright_{I_E \cup I_S}$, then all the following are true: $UIA(SYS) \trianglelefteq UOA(TM)$, $UOA(SYS) \trianglelefteq UIA(TM)$, $TRC(SYS) \trianglelefteq TRC(TM)$, $EIN(SYS) \trianglelefteq EON(TM)$, $EON(SYS) \trianglelefteq EIN(TM)$, $PT(SYS) \trianglelefteq PT(TM)$, $PI(SYS) \trianglelefteq (PI(TM) \setminus dDIV(TM))$ and $dDIV(TM) \subseteq PI(TM)$, where $dDIV(TM) = \{t \mid \exists s : s^0 \stackrel{t}{\Longrightarrow} s \bullet s \in Q\} \cdot \{d^\omega\}$.*

*Proof.* Follows from Definition 5.8. □

**Proposition 5.11.** *Given PROT, $ENV(PROT)$ is the least refined (i.e. worst) environment conforming to PROT, and $SYS(PROT)$ is the least refined system conforming to PROT.*

*Proof.* First, it is easy to prove $ENV(PROT)$ and $SYS(PROT)$ both conform to $PROT$. Then, let us prove that $SYS(PROT)$ (with alphabet $A = I_S \cup I_E$) is the least refined conforming one.

W.o.l.g. assume a GLAC in normal form, $GLAC \mathrel{\widehat{=}} NC \setminus \Delta$ where $NC \mathrel{\widehat{=}} \|[\vec{GLAG}]$, is worse than $SYS(PROT)$ but $ENV(PROT)[GLAC]$ is DCIR-free. Then there must be a compatible environment $E[\cdot]$ such that $E[SYS]$ is DCIR-free while $E[GLAC]$ is not. W.o.l.g. assume the environment is also in normal form, i.e. $E[\cdot] \mathrel{\widehat{=}} (\|[\vec{GLAG'}] \parallel [\cdot]) \setminus \Delta'$, and $\Delta$ is disjoint from the alphabet $A'$ of $\|[\vec{GLAG'}] \parallel [\cdot]$. Therefore, $\|[\vec{GLAG'}] \parallel (NC \setminus \Delta) \setminus \Delta'$ is the same as $CC \setminus (\Delta \cup \Delta')$ where $CC \mathrel{\widehat{=}} \|[\vec{GLAG'}] \parallel NC$.

In case $E[GLAC]$ is not free of relative starvations, assume $\bar{t}$ is one of its $CC$'s traces causing relative starvations. Then $\bar{t} \upharpoonright_{A'}$ must be infinite (due to divergence-freedom of $E[GLAC]$), $\bar{t} \upharpoonright_A$ must be in $TRC(SYS)$ and $\bar{t} \upharpoonright_{A \cup \Delta}$ must be in $TRC(NC)$ (the same argument as before). These imply that the relative starvations should be on and only on $\vec{GLAG}$ side (otherwise, $E(SYS)$ will have similar relative starvations). $\bar{t} \upharpoonright_A$ cannot be infinite (and in $PI(SYS) \subseteq PI(ENV)$), since that implies $ENV(GLAC)$ has similar relative starvations. However, if $\bar{t} \upharpoonright_A$ is finite, then it should be in $PT(SYS) \cup EON(SYS)$ (otherwise being in $EIN(SYS)$ implies $E[SYS]$ will have relative starvations.) Thus $(\bar{t} \upharpoonright_A)d^\omega$ should be in $PI(ENV)$, which will cause relative starvations in $ENV(GLAC)$ (on $\vec{GLAG}$). Contradiction. Hence, we can conclude that $SYS(PROT)$ is the least refined system conforming to $PROT$.

Similarly, in case $E[GLAC]$ is not free of divergences, chokes or illegal stops, contradiction can be found. Dually, the least refinedness of $ENV(PROT)$ can be proved. □

**Theorem 5.12.** *Assume a vector of protocols $\vec{PROT}$, a generalised context $GC[\vec{x}]$, and a protocol $PROT'$, then $ENV(PROT')[GC[SYS(\vec{PROT})]]$ is free of DCIRs implies $\forall \vec{GLAC} \bullet (\vec{GLAC}$ conforms to $\vec{PROT} \Rightarrow GC[\vec{GLAC}]$ conforms to $PROT')$, and vice versa.*

*Proof.* Follows from Proposition 5.11. □

## 6 An example

Let us give an example (2-level hierarchy) to illustrate how we use schedulers and protocol conformance to verify real-world circuits. The system is a gate-level circuit that implements the *FalseVariable* component in Balsa [5] and we only consider safeness (i.e. choke-freedom).

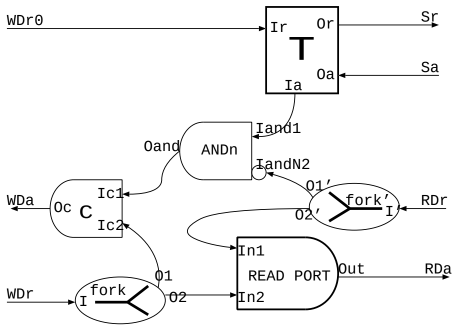


*Figure 1. FalseVariable (FV) implementation*

The root level node (the FV system) has the protocol (i.e. $PROT(FV)$) specified using the STG (Signal Transition Graph) [2, 17] in Fig 2(a). It is also the specification that the FV system needs to satisfy. Translating the STG in Fig 2(a), we obtain:

$$\begin{array}{l} protocolFV = (writer \mathbin{|||} reader);\ WDa.down;\ protocolFV \\ writer = (WDr.up \mathbin{|||} RDr.down);\ WDa.up;\ WDr.down \\ reader = (WDr0.up;\ Sr.up;\ RDr.up \mathbin{|||} WDr.up);\ RDa.up \\ \quad ;\ Sa.up;\ Sr.down;\ RDr.down;\ RDa.down;\ Sa.down \end{array}$$

$|||$ is an interleaving operator, while ; is the sequential composition operator[7]. Sequential composition binds stronger

[6] *SYS* and *TM* are used as the abbreviations of $SYS(PROT)$ and $TM(PROT)$.

[7] To be consistent with theoretical CSP [12], $a;\ a'$ should be written as $a \to Skip;\ a' \to Skip$. But for some CSP variants in asynchronous circuit community, *Skip* can be omitted. For this example, we follow the latter approach.

than choices, while choices are stronger than parallel composition.

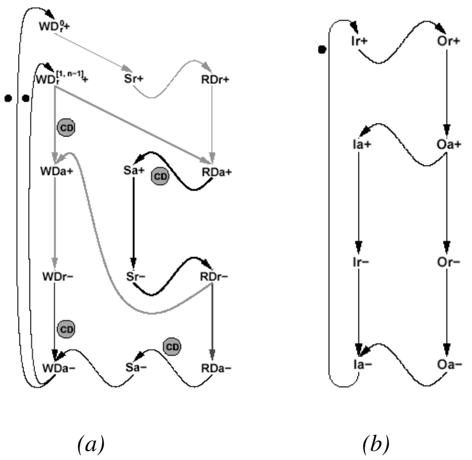


*Figure 2. The gate-level protocol of FV and T components*

At the leaf level, besides classic gate elements such as *AND*, Muller-C, fork, etc., there is also a special $T$ component and a *READPORT* component in the circuit. We do not care about the internal implementation of the $T$ component or the *READPORT* component (i.e. $LAC(T)$ and $LAC(READPORT)$, which might add more levels to the hierarchy). But $T$'s protocol is given using the STG in Fig 2(b), and *READPORT* (with abstraction) works like an *AND* gate. We simply assume the two components conform to their protocols, whose CSP translation are:

$$protocolT = Ir.up;\ Or.up;\ Oa.up;$$
$$(Or.down;\ Oa.down \ ||| \ Ia.up;\ Ir.down);\ Ia.down;\ protocolT$$

$$readport = (In1.up \ ||| \ In2.up);\ Out.up;\ readport'$$
$$readport' = (\ In1.down;\ (Out.down \ ||| \ In2.down)$$
$$\Box\ In2.down;\ (Out.down \ ||| \ In1.down)\ )$$
$$;\ readport$$

For other leaf nodes, although it is possible to specify the components' full behaviour ($LAG(X)$), it is complicated to do so and the results may be hard to justify without resorting to low-level physical arguments (interested readers can try to specify the full behaviour of the *AND* gate.). Sometimes, it is much easier to directly give its protocol which are simpler, and our intuition will suffice to justify.

There are two forks in the circuit; their protocol is:

$$fork0 = I.up;\ (O1.up \ ||| \ O2.up);\ fork1$$
$$fork1 = I.down;\ (O1.down \ ||| \ O2.down);\ fork0$$

The Muller-C element has the protocol:

$$protocolC = (Ic1.up \ ||| \ Ic2.up);\ O.up;\ protocolC'$$
$$protocolC' = (Ic1.down \ ||| \ Ic2.down);\ O.down;\ protocolC$$

The protocol of the one-input negated *AND* gate is very different from that of *READPORT*, though they both 'are' *AND* elements. This is a good example of verification using protocols instead of the full specification of elements.

$$andN = IandN2.up;\ (Iand1.up \ ||| \ IandN2.down)$$
$$;\ Oand.up;\ andN'$$
$$andN' = Iand1.down;\ Oand.down;\ andN$$

For all these elements, our intuition can easily justify that the components ($LAG(X)$) conform to their protocols ($PROT(X)$). Therefore, to verify the FV system against its specification, that is,

$$LAG(C) \parallel LAG(fork) \parallel LAG(READPORT) \parallel LAG(fork') \parallel$$
$$LAG(ANDn) \parallel LAG(T) \text{ conform to } PROT(FV)$$

we only need to show, using Theorem 5.12 and Definition 5.3, that

$$SYS(PROT(C)) \parallel SYS(PROT(fork)) \parallel$$
$$SYS(PROT(READPORT)) \parallel SYS(PROT(fork')) \parallel$$
$$SYS(PROT(ANDn)) \parallel SYS(PROT(T)) \parallel TM(PROT(FV))$$

is free of chokes.

*TM* and *SYS* of a protocol above give exactly the same process as that of the protocol. The only change is on the alphabets, i.e. the wire connections, which will be embodied in the specification of the scheduler. Thus, the above instrumentation system can be implemented as the CSP system below.

$$test\_system = scheduler \parallel protocolFV \parallel protocolC$$
$$\parallel fork0 \parallel readport \parallel fork0' \parallel andN0 \parallel protocolT$$

The specification of the scheduler is as below:

$$CN = \{(WDr0, Ir), (Oc, WDa), (WDr, I), (O1, Ic2),$$
$$(O2, In2), (O1', IandN2), (O2', In1), (Oand, Ic1),$$
$$(Ia, Iand1), (Or, Sr), (Sa, Ia), (RDr, I')\}$$

$$scheduler(CN) =$$
$$\Box\, x : \mathrm{dom}\, CN \bullet x?z \to CN(x)!z \to scheduler(CN)$$

where $CN$ is the function encoding wire connections.

It is easy to see *test_system* never terminates. Thus all the deadlocks in the system are caused by chokes. Checking deadlock freedom on *test_system* with FDR2 [6], we find deadlocks. The deadlock trace, thus the choke trace, is:

$$(WDr.up, I.up)\ (O2.up, In2.up)\ (WDr0.up, Ir.up)$$
$$(Or.up, Sr.up)\ (RDr.up, I'.up)\ (O2'.up, In1.up)$$
$$(Out.up, RDa.up)\ (Sa.up, Oa.up)\ Ia.up$$

This example is for illustration purpose only. More case studies are needed for the progress conditions. To handle relative starvation correctly, FDR2 will need some small adaptation to capture all the divergences in a process, which was claimed to be no problem by Roscoe [13].

## 7 Conclusion and future work

We have proposed a compositional verification theory for asynchronous concurrent systems, integratable into standard process algebra theories. More specifically,

- Our theory extends XDI and RPT theories to infinite traces and enables us to capture a new class of errors, relative starvations, undetectable in previous models. Our proof of the compositionality theorem is purely set-theoretical.
- We formulate our theory in a way that is amenable to direct translation to process algebra, and, with the help of the newly introduced infinite traces model of CSP, $\mathcal{SBD}$, we give a formal translation from these systems to a major process algebraic language, CSP, and show their semantics correspondence (up to isomorphism).
- We reduce asynchronous circuit verification problems to CSP refinement checking problems, and prove their correctness. This gives us an immediate opportunity to exploit the power of the process algebra model-checking tools, e.g. FDR2.

From a practical point of view, the scheduler approach is natural to use, and the extra states introduced by a scheduler is negligible in verification. The real problem in asynchronous circuits causing state space explosion is unnecessary interleaving. With FDR2, we have shown scalability in verifying some classical asynchronous circuit problems. For instance, using *chase* compression, the verification time of the tree arbiter example becomes linear in the size of the tree and we manage to verify tree arbiters up to $2^{10}$ ways [16]. More work is underway to design more advanced state space reduction techniques inspired by asynchronous circuits for integration into FDR2.

**Acknowledgements** This paper benefits from informal discussions with M.B. Josephs. We thank A.W. Roscoe and M. Goldsmith for pointing us to their recent works and help with FDR2. Invaluable advice and help from the members of the AMULET group, and in particular Doug Edwards and Andrew Bardsley, are also greatly appreciated. The research is funded by EPSRC projects GR/S11091/01 & GR/S11084/01.

## References

[1] K. van Berkel. *Handshake circuits - an Asynchronous Architecture for VLSI Programming*. CUP, 1993.

[2] T. A. Chu. Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications. PhD thesis, MIT, 1987.

[3] A. Davis and S. M. Norwick. *An Introduction to Asynchronous Circuit Design*. The Encyclopedia of Computer Science and Technology (vol 38), Marcel Dekker, 1998.

[4] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. MIT Press, 1988.

[5] D. Edwards and A. Bardsley. Balsa: An Asynchronous Hardware System. Principles of Asynchronous circuit Design, Part II.

[6] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, 1999.

[7] J. He and K. J. Turner. Verifying and Testing Asynchronous Circuits using LOTOS. FORTE XIII/PSTV XX, pages 267-283, 2000.

[8] M. B. Josephs. An Analysis of Determinacy Using a Trace-Theoretic Model of Asynchronous Circuits. ASYNC 2003: 121-131

[9] M. B. Josephs. Receptive Process Theory. Acta Informatica 29(1):17-31, 1992.

[10] H. K. Kapoor and M. B. Josephs. Modelling and verification of delay-insensitive circuits using CCS and the concurrency workbench. Information Processing Letters, v.89 n.6, p.293-296, 31 March 2004.

[11] Y. Liu. Amulet1: Specification and verification in CCS. PHD thesis, University of Calgary, Canada, 1996

[12] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[13] A. W. Roscoe. Seeing beyond divergence. Proceedings of "Symposium on the Occasion of 25 years of CSP", London, July 2004, LNCS 3525.

[14] K. Stevens. Practical Verification And Synthesis Of Low Latency Asynchronous Systems. PhD Thesis, University Of Calgary, 1994.

[15] T. Verhoeff. Analyzing Specifications for Delay-Insensitive Circuits. ASYNC 1998: 172-183

[16] X. Wang, M. Kwiatkowska, G. Theodoropoulos and Q. Zhang. Opportunities and challenges in process-algebraic verification of asynchronous circuit designs. To appear in ENTCS, FMGALS'05, July, 2005.

[17] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno and M. Pietkiewicz-Koutny. On the Models for Asynchronous Circuit Behaviour with OR Causality. FMSD Vol.9, No.3, Nov.1996, pp. 189-234.