Fundamenta Informaticae XXX (200?) 0–27
IOS Press

On process-algebraic verification of asynchronous circuits[*]

Xu Wang[†]
International Institute of Software Technology
United Nations University
P.O. Box 3058, Macau
wx@iist.unu.edu

Marta Kwiatkowska[‡]
School of Computer Science
University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
M.Z.Kwiatkowska@cs.bham.ac.uk

Abstract. Asynchronous circuits have received much attention recently due to their potential for energy savings. Process algebras have been extensively used in the modelling, analysis and synthesis of asynchronous circuits. This paper develops a theoretical basis for using process algebra and associated model checking tools to verify asynchronous circuits. We formulate a model that extends 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. Finally, the correspondence of our theory with trace theory (i.e. prefix-closed trace structures), receptive process theory and the XDI model is investigated.

Keywords: Process algebra, asynchronous circuits, assume/guarantee verification, progress/liveness, and CSP/FDR2.

[*]A preliminary version of this paper was presented at the ACSD 2006 conference [28].
[†]The work was carried out while the author was affiliated with Birmingham University.
[‡]

1. Introduction

Asynchronous hardware designs have received much attention recently due to their potential for energy savings, and consequently their suitability for deployment in embedded and mobile devices such as smartcards. Fundamental differences of timing in asynchronous hardware render conventional hardware description languages inappropriate for their modelling and make the design susceptible to deadlock and race conditions. Verification of asynchronous hardware is therefore an important area of study.

Asynchronous circuits are usually modelled using asynchronous concurrent systems, a fundamental semantic model for concurrency formalisms such as process algebras and Petri nets based on labelled transition systems [22]. Classically, asynchrony has two meanings: clock asynchrony (versus lock-step synchrony) and communication asynchrony (versus rendez-vous). Asynchronous circuits contain both these features. At the level of executions, concurrency can be modelled as interleaving concurrency or true concurrency, with the latter making finer distinctions about *causality* of events. True concurrency and related formalisms [15, 22, 30] are widely considered as the most appropriate for modelling of asynchronous circuits. Petri nets [16, 21, 12] and Signal Transition Graphs (STGs) [2, 31, 3] are frequently used. Software tools such as Petrify [31] exist, which support the analysis of safety and liveness, and also increasingly automatic synthesis [3] of asynchronous circuits. Such graph-based models, however, are not compositional, and therefore have no native support for componentwise reasoning about correctness.

Process algebras, on the other hand, are compositional. In the past they have been used, together with associated verification tools, for the modelling and verification of asynchronous circuits; we mention CCS [13, 23], LOTOS [8], and CSP [11, 26]. An important advantage of process algebras is that they are not only good at modelling asynchronous circuits in detail, but can also describe the function of the circuits at a higher level of abstraction. Thus, we can program a circuit in a high-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]), called *silicon compilation*. Thus, process algebras support hierarchical description that is frequently employed in large-scale industrial designs.

This paper is part of an effort to formulate a rigorous approach for the verification of asynchronous circuit designs developed using the Balsa asynchronous circuit synthesis system [6]. Balsa (and similarly Tangram) use as their high-level language some variant of CSP. Thus, conventional CSP theory suffices for the high-level verification, i.e. for functional correctness, and the FDR2 model checker [7] for CSP can be applied at that level in a straightforward way [26]. At the lower level, asynchronous circuits are usually modelled and analysed with the help of tailored asynchronous verification theories, such as trace theory [5], receptive processes [10] or XDI theory [24]. To tackle the state-space explosion problem, asynchronous verification theories [5, 24] usually adopt a compositional approach (i.e. in the style of assume/guarantee). This is possible largely because, at the lower level, the communication is asymmetric (input/output distinction) and non-blocking, which is not the case for most process algebras.

In this paper, we formulate a compositional verification theory for asynchronous circuits, called *protocol conformance verification*, and embed it in a standard process algebra theory such as that of CSP and its semantic models. This embedding enables componentwise automatic verification in the style of assume/guarantee reasoning using the FDR2 model checker. Our model is based on the intuition of the XDI model and Receptive Process Theory (RPT) [24, 9, 10], but extends previously known results by addressing advanced progress conditions (*relative starvations*). One advantage of our approach is that it avoids the complexity of Dill's Complete Trace Structure construction [5].

In the rest of the paper, we specifically target CSP [19], but many of the results also apply to other process algebras. First, in Section 2, 3, and 4, we give a general labelled transition system (LTS) 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 faithfully translated into CSP using an infinite traces model of CSP [20] and a simple but novel idea of *a scheduler*. The construction of the scheduler (see examples in Section 6) helps to avoid the need to impose process receptiveness and results in 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 thus a type of local livelock. 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, which are automatic, and prove the correctness of our approach.

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 compositionality proof is accomplished purely using set-theoretical techniques. The whole approach is illustrated by a real-world case study of the False Variable circuit (Section 6). Finally, in Section 7, the correspondence of our theory with trace theory (i.e. prefix-closed trace structures), receptive process theory and XDI is investigated.

2. Asynchronous concurrent systems

In this section we introduce our model of asynchronous concurrent systems, which is based on labelled transition systems (LTS). The model represents the behaviour of circuits built from logical gates in a compositional fashion. The meaning of this is made precise in Section 7, where we demonstrate that our theory coincides with and extends [5, 9, 10, 24]. Therefore, it supports asynchronous circuit verification which typically involves checking of safeness and progress conditions. We do not model true concurrency in the sense of independence relation, but nevertheless make fine causality distinctions.

The definition of the model is as follows. An *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, $S \cup \{Choke\}$, where $Choke \notin S$.
- A transition relation $T : S \times (A \cup \{\tau\}) \leftrightarrow (S \cup \{Choke\})$. τ 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.

Note that we make a distinction between input and output actions, and also distinguish a silent (invisible) action τ. In the rest of the paper, $(\!|\,|\!)$ is the image operation on relations. Let e denote a non-τ event,

a denote a τ or non-τ event, and Δ denote a subset of A. t and u are finite traces, i.e. finite sequences of non-τ events (including the empty sequence, ϵ)[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.

Next, we define the following constructs.

Definition 2.1. Given an IOTS (I, O, S, T, s^0) and its states 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 $ETS = ETS^I \cup ETS^O$.
- $s \longrightarrow$ iff there exists a state s' and event a in the IOTS such that $s \xrightarrow{a} s'$. When $\neg(s \longrightarrow)$, 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 path is a finite sequence of alternating states and events, $s_0 a_1 s_1 a_2 ... a_n s_n$ ($\in S \times (A^\tau \times S)^*$), where $(s_{i-1}, a_i, s_i) \in T$ for all $1 \le i \le n$. A loop is a path whose starting and ending states are the same. The number of steps in a path is the length of the event sequence contained in the path. A path consisting of a single state, e.g. s, is a *zero step path*.
- A state s is *reachable* iff there is a zero or nonzero step path from s^0 to s.
- A state s is *stable*, i.e. *stable*(s), iff $\neg(s \xrightarrow{\tau})$.
- A state s is *divergent*, i.e. *divergent*(s), iff there is a zero or nonzero step τ-path going from s to a state s' such that s' is in a τ-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 τ-path going from s to s'.
- $s \overset{e}{\Longrightarrow} s'$ iff there exist s_0 and s_1 such that $s_0 \xrightarrow{e} s_1$, $s \overset{\epsilon}{\Longrightarrow} s_0$ and $s_1 \overset{\epsilon}{\Longrightarrow} s'$.
- $s \overset{t}{\Longrightarrow} s'$, where $t = et_0$, 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 maintaing the order. It gives us another trace in A^*. $\restriction_A$ can be lifted to operate on infinite traces or sets of traces.

[1]Sometimes a and e are also used to denote singleton sequence and singleton trace.

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

An asynchronous circuit, modelled as an IOTS, is regarded as a hierarchical system of modules, composed from logical gates using process-algebraic operators such as parallel composition.

Formally, a *LAG* (*Logical Abstract Gate*) is a τ*-free and choke-free IOTS*. This ensures that we can use any formalism (e.g. Petri net [2, 31] or process algebra [19]) to specify it, as long as the formalism can be given an LTS semantics. From LAGs, a *LAC* (*Logical Abstract Circuit*) can be built using hiding and parallel operators:

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

In the above, Δ_O is the set of actions to be hidden and $\parallel$ is the alphabetised parallel operator. 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 such *non-interfering* LACs, i.e. $O(LAC) \cap O(LAC') = \{\}$. On the other hand, components are allowed to share inputs (i.e. broadcast communication in an isochronic 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 is called 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 $O = O_1 \cup O_2$, $I = (I_1 \cup I_2) \setminus O$, $A = A_1 \cup A_2$, $S = S_1 \times S_2$,

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

$T : S \times A \cup \{\tau\} \leftrightarrow S$ 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:

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

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

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 operator relabels every to-be-hidden event in an LTS to τ, just like in CSP.

Assume an infinite set of LAC variables (typed by their input and output 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 $C[\cdot]$.

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 $\vec{x}$.

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 $SC[\vec{LAG}]$.

Theorem 2.1. ([5])
Given $SC[\vec{LAG}]$, it has a normal form, $(\parallel[\vec{LAG'}]) \setminus \Delta$, where $\vec{LAG}$ and $\vec{LAG'}$ are of the same dimension and $\parallel$ 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. and that $SC[\vec{LAG}]$ and $(\parallel[\vec{LAG'}]) \setminus \Delta$ are isomorphic.

Given $IOTS$ and $IOTS'$, we say that they are renaming-isomorphic iff there is a 1-to-1 renaming[2] on one of them such that the resulting IOTS is isomorphic to the other.

3. The CSP model

In this paper we assume familiarity with CSP [19]. Recall that the main CSP operators are: $a \rightarrow P$ (prefix) , $\Box$ (external choice), $\sqcap$ (internal choice), $|[A]|$ (interface parallel), $\parallel$ (alphabetised parallel), $|||$ (interleaving parallel), $\mu\ l.F(l)$ (recursion), *Skip* and *Stop*. Appendix A summarises the CSP operators used in this paper.

Stable failures and failure/divergences are the main semantic models of CSP [19] used in this section. They are both finite trace models. However, we have also utilised a newly developed infinite trace CSP model [20], the $\mathcal{SBDF}$ 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. $\mathcal{SBDF}$ 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 type of CSP deadlock, thus avoiding the need to impose receptiveness requirement on processes.

The basic idea is that *all events* (including τ) are split into *two* events, e.g. e into e^i and e^o, and τ into τ^i and τ^o [3]. e^o (or τ^o) is used by the sending side (one LAG) of e while e^i and τ^i by the receiving side (one or more LAGs).

[2]A renaming is 1-to-1 iff it will not make any two different events become the same event.
[3]Note that, unlike τ, τ^i and τ^o should be treated as normal events.

For instance, given $LAG = (I, O, S, T, s^0)$, it can be translated into CSP via the mapping:

$$CSP(LAG) \mathrel{\widehat{=}} P(s^0)^4,$$

where

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

Then, a complete circuit, *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:

$$scheduler(X) = \ \Box\, a : X \bullet a^o \rightarrow a^i \rightarrow scheduler(X).$$

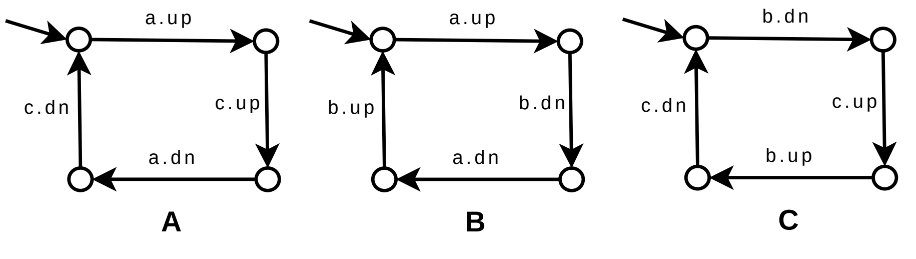
Note that only complete LACs need to be translated since the verification theory introduced in Section 5 will reduce any verification problem to checks on some complete LAC. The scheduler synchronizes with all the signal transitions in *LAC* to control their occurrences. It first checks the signal transitions available for sending in the current state, 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 is reached when selecting a^o, this implies termination of the circuit execution.

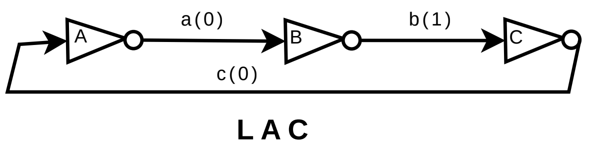
The translation of *LAC* is given by:

$$CSP(LAC_1 \parallel LAC_2) = CSP(LAC_1) \,|[I_1^i \cap I_2^i]|\, CSP(LAC_2)$$
$$CSP(LAC \setminus \Delta) = (CSP(LAC) \,|[\Delta^o \cup \Delta^i]|\, scheduler(\Delta))[e : \Delta \bullet e^i \leftarrow \tau^i, e^o \leftarrow \tau^o]$$

In the above, $|[\Delta]|$ is the (binary) interface parallel operator [19]. Its two arguments synchronize on, and only on, events in Δ. 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 τ^i and τ^o (which are visible).

Example 3.1. Let us consider a simple example of *LAC* (see diagram below) consisting of three LAGs (here inverters), *A*, *B*, and *C*, which are connected by three wires, *a*, *b*, and *c*. More specifically, $LAC = (A \parallel B) \setminus \{a\} \parallel C$.





^{4}We assume here that recursion will not introduce extra τ events.

Then, by definition of the translation mapping we derive that $CSP(A)$ is:

$$PA = a.up^o \rightarrow c.up^i \rightarrow a.dn^o \rightarrow c.dn^i \rightarrow PA$$

since a is A's output and c is A's input. Similarly, $CSP(B)$ and $CSP(C)$ are:

$$PB = a.up^o \rightarrow b.dn^i \rightarrow a.dn^o \rightarrow b.up^i \rightarrow PB$$
$$PC = b.dn^o \rightarrow c.up^i \rightarrow b.up^o \rightarrow c.dn^i \rightarrow PC$$

Next, $CSP((A \parallel B) \setminus \{a\})$ is $SUB = ((PA \,[\![\, \{\} \,]\!]\, PB) \,[\![\, \{a^i, a^o\} \,]\!]\, scheduler(\{a\}))[a^i \leftarrow \tau^i, o^o \leftarrow \tau^o]$. Note that PA and PB are composed through parallel but synchronise on an empty set since A and B share no common input. The local scheduler $scheduler(\{a\})$ will wait on a^o from PA and force a^i onto PB. The local scheduler and the renaming of a^o and a^i to τ^o and τ^i implement the hiding on $A \parallel B$.

Finally, $CSP(LAC) = SUB \,[\![\, \{\} \,]\!]\, PC$ and $\Omega(LAC) = (SUB \,[\![\, \{\} \,]\!]\, PC) \,[\![\, \{b^i, b^o, c^i, c^o, \tau^i, \tau^o\} \,]\!]\, scheduler(\{b, c, \tau\})$. Note that LAC is a complete circuit and its set of outputs is $\{b, c\}$, since a is hidden.

Splitting events and introducing a scheduler may seem unnatural. However, it should be viewed, instead, in the context of real circuit applications, where an event is usually associated with a wire. A wire connects two components, where each component has a different local name for the wire (one for input and another for output). When specifying a protocol, it is good practice to use the local names rather than the global name, since this implies 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 a very natural solution. Section 6 illustrates how one can use schedulers to verify choke-freedom of an asynchronous circuit.

Definition 3.1. Given a complete LAC and its translation $\Omega(LAC)$, 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 3.1. Given a complete LAC and its translation $\Omega(LAC)$, all transient states have exactly one incoming transition and one or zero outgoing transitions.

Proof:
Induction on the structure of LAC. □

Proposition 3.2. Given a complete LAC and its translation $\Omega(LAC)$, 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. □

Proposition 3.3. Given a complete LAC and its translation $\Omega(LAC)$, 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:

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

- 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:
Since *LAC* can be written as $SC[\vec{LAG}]$, the set of states of *LAC* is the cartesian product of the sets of states of $\vec{LAG}$. Since our translation to CSP preserves the process structure, $\Omega(SC[\vec{LAG}])$ is a network of $CSP(\vec{LAG})$ combined with the scheduler, with the LTS of each $CSP(LAG)$ isomorphic to that of *LAG*. The scheduler has two states. It is not difficult to see that each state $\vec{s}$ of $SC[\vec{LAG}]$ is related to two states in $\Omega(SC[\vec{LAG}])$. □

Theorem 3.1. Given a *LAC* and its translation $\Omega(LAC)$, *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

$$goodsched(O) = \; Stop \sqcap \; \Box a : O \cup \{\tau\} \bullet (a^o \rightarrow a^i \rightarrow goodsched(O)).$$

Proof:
Use Proposition 3.3. □

Theorem 3.2. Given a *LAC* and its translation $\Omega(LAC)$, *LAC* is divergence-free iff $\Omega(LAC) \setminus \{\tau^i, \tau^o\}$ is divergence-free in the failure/divergences model.

Proof:
Use Proposition 3.3. □

4. Safeness and progress conditions

Compared to synchronous circuits, the design of asynchronous circuits faces a major challenge, namely *glitches* [4]. A glitch happens when there exists an assignment of delays that can cause an unwanted signal transition to occur in the circuit. A phenomenon known as 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 corresponds to *safeness*, one of the main correctness requirements of asynchronous circuits.

However, the above framework is inadequate for capturing 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 [24], 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 models that incorporate LAGs and LACs. A *GLAG* (*Generalised Logical Abstract Gate*) is:

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

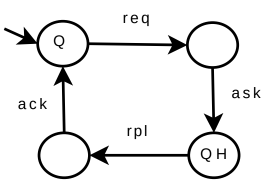
where Q ($\subseteq S$) labels the set of *quiescent* states of LAG, and QH ($\subseteq Q$) labels the set of *quiescent hunger* states. A state of LAG is quiescent iff it *may* stop to produce output. A state is a hunger state iff it *may* need progress requirement on the environment. A state is a quiescent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

Labelling 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.

First, it is necessary to add well-formedness conditions by requiring:

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

Example 4.1. For example, with $I = \{req, rpl\}$, $O = \{ack, ask\}$, and Q and QH states marked, the LTS below describes a GLAG that, when requested by the environment, will ask for help from a helper. After getting the helper's reply it sends back acknowledgement to the environment.



The initial state (marked by Q) is a quiescent state because it is waiting for an input that may or may not arrive. The state marked by QH is a quiescent hunger state because the input it is waiting for must eventually arrive.

A GLAG defines a finite-state labelled transition system. A GLAG with a deterministic LTS is called *transparent*, e.g. the above example.

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

A GLAG with a nondeterministic LTS can be transformed to a transparent GLAG using a determinization operation, DT, similar to the subset construction.

Definition 4.2. Given *GLAG*, $DT(GLAG)$ constructs another GLAG, $GLAG'$, such that

- $I' = I$, $O' = O$, $S' = \mathbb{P}(S)$, $s'^0 = \{s^0\}$
- $Q' = \{s' \mid \exists s \in s' \bullet s \in Q\}$ 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 $ets(s') = \{o : O \mid \exists s \in s' \bullet o \in ETS(s)\} \cup \{i : I \mid \forall s \in s' \bullet i \in ETS(s)\}$.

Like subset construction, a state of $GLAG'$ is denoted by a subset of states of *GLAG* and R_e is a relation between states that are connected by e-labelled transition in *GLAG*. Unlike subset construction, at a state s' of $GLAG'$ an input is enabled iff the input is enabled at every element of s' in *GLAG*; an output is enabled iff the output is enabled in *GLAG* at at least one element of s'. This is defined in the enabledness function *ets*.

DT can be lifted to operate on a vector of GLAGs. We can now state an important proposition about *DT*; other propositions about this operation can be found in Section 5.

Proposition 4.1. *DT* is closed on the well-formed GLAGs.

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

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

The definitions of super-combinators and transition rules of LACs can be carried over to those of GLACs.

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 4.3. 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 on the circuit behaviour, which becomes obvious if we look at its normal form, $(\|[\vec{GLAG'}]) \setminus \Delta$. $A' \setminus \Delta$ defines the point of view of $SC[\vec{GLAG}]$ (here, A' is the alphabet of $\|[\vec{GLAG'}]$). By changing the subset Δ, 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 4.4. 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 4.5. A complete circuit $SC[\vec{GLAG}]$ is *free of relative starvations* iff there is no relative starvation on any variable 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 τ^i and τ^o, they are just normal events (not like $\surd$ 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 Ω, to complete GLACs as follows (only the modified rules are listed):

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

$$CSP(GLAC_1 \parallel GLAC_2) = CSP(GLAC_1) \mathbin{[\![} (I_1^i \cap I_2^i) \cup \{\surd^i, \surd^o\} \mathbin{]\!]} CSP(GLAC_2)$$

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

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

$P(s)$ is modified 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 3.2 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 the following.

Theorem 4.1. Given $GLAC$ and its translation $\Omega(GLAC)$, $GLAC$ 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

$$goodsched(O) = \quad Stop \sqcap \Box\, a : O \cup \{\tau, \surd\} \bullet (a^o \rightarrow a^i \rightarrow goodsched(O)).$$

Proof:
Use Proposition 3.3 which can be largely carried over to the new Ω, except for the part related to $\surd$.

- From $GLAC$ to $\Omega(GLAC)$. For any quiescent hunger state s in $GLAC$, all the GLAGs in $GLAC$ will also be in quiescent states. From the translation above, the non-transient state $t = f(s)$ in $\Omega(GLAC)$ should have a transition labeled by $\surd^o$ to one transient state, say t'. Since at state s there is at least one GLAG, say $GLAG$, in quiescent hunger states and not having $\surd^i$ transition in $\Omega(GLAG)$, it disables the transition in $\Omega(GLAC)$ due to the multiway synchronization on $\surd^i$. Hence, t' is a sink state.

- From $\Omega(GLAC)$ to $GLAC$. If a non-transient state t can transit to a state t' using $\surd^o$, it implies all the GLAGs in $GLAC$ are in the quiescent state due to multiway synchronization of CSP on $\surd^o$. So $f^{-1}(t)$ is a quiescent state of $GLAC$. Since t' is a sink state, it implies that at state $f^{-1}(t)$ there is at least one GLAG in quiescent hunger states. Hence, $f^{-1}(t)$ is a quiescent hunger state in $GLAC$.

$\square$

Theorem 4.2. 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

$$K(DT(GLAG_x)) \sqsubseteq_{SBDF} \Omega(\|[\vec{GLAG}] \setminus (A \setminus A_x)) \setminus \{\tau^i, \tau^o, \surd^o, \surd^i\},$$

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

$$\begin{aligned} k(s) = {} & \Box(e, s') : \{(e, s') \mid s \xrightarrow{e} s'\} \bullet e^o \rightarrow e^i \rightarrow k(s') \\ & \sqcap \text{ if } s \in QH \text{ then } Stop \text{ else } (\mu\, l.(l) \sqcap Stop).^6 \end{aligned}$$

Proof:
Recall that, in the $\mathcal{SBDF}$ model, a process is denoted by (F, I, D) and refinement is component-wise supersethood. As the process on the RHS of refinement is free of chokes and illegal stops, the traces (including the infinite traces I) of $K(DT(GLAG_x))$ should be a superset of that of the RHS (due to the fact that all the traces removed after the DT operation have a prefix leading to *Choke* state, c.f. the enabledness function *ets* in Definition 4.2). In any state s of $DT(GLAG_x)$, no matter how the conditional (if $s \in QH$) evalutes, *Stop* (maximal failures) will be associated with the state by internal choice. That means the failures set of $K(DT(GLAG_x))$ will be a superset of the RHS. If D of $K(DT(GLAG_x))$ is also a superset of the RHS, then the refinement will hold. Since $K(DT(GLAG_x))$ diverges (i.e. $\mu\, l.(l)$) on and only on the traces that cannot reach $GLAG_x$'s QH state (due to the fact that the DT operation will not change the set of traces that can reach QH), we have D of $K(GLAG_x)$ contains D of the RHS iff the RHS does not diverge on $GLAG_x$'s quiescent hunger states. $\square$

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. We associate each of them with a *protocol*.

A protocol can be defined as an arbitrator positioned 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:

[6] Recursion $\mu\, l.(l)$ defines a single state process having only a τ self-loop, i.e. pure divergence.

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

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:

- As an assume/guarantee approach to specifying the behaviour of components: a circuit component produces correct outputs so long as the environment provides legal inputs. This is 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 a d-labelled self-loop inserted at all Q states.

$TM(PROT)$ derives a testing machines from given *PROT*. Relative to four undesirable observables, i.e. divergences, chokes, illegal stops and relative starvations (referred to DCIRs), protocol conformance is defined using the correct instrumentation of a component by *ENV* and *SYS* as follows.

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 the definition of protocol and conformance, a hierarchical system, represented as a n-level (syntax) 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 of leaf nodes, we use its protocol to derive *ENV* and instrument the parallel composition of the sons' protocols. If correct, it implies that the component at level 2 conforms to its protocol. This algorithm proceeds until it reaches the root level (i.e. level n). Therefore, we can break 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 5.1 at the end of this section.

However, before we can state the proof of this theorem, we first need to define a refinement (i.e. better than) relation ($\preceq_S$) between components. The idea is that Theorem 5.1 can be derived from the fact that *ENV* and *SYS* are, respectively, the worst environment and system conforming to a protocol (i.e. Proposition 5.5).

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 5.1. $\preceq_S$ 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 5.2. $\preceq_S$ 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 $E[C[GLAC]] \preceq_S E[C[GLAC']]$. □

We are now ready to prove the main theorem. Our proof will be set-theoretical, and will use the following partitioning of the (finite and infinite) trace sets.

Definition 5.6. Given a GLAC in a restricted form (i.e. without hiding), denoted $NC \mathrel{\widehat{=}} \|[\overrightarrow{GLAG}]$, 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:

$$UIA = \{t' \mid \exists\, t, u : A^* \,\exists\, s : S \,\exists\, i : I \bullet s^0 \overset{t}{\Longrightarrow} s \wedge t' = tiu \wedge \neg\, (s \ \overset{i}{\rightarrow}\ )\}$$
$$UOA = \{t' \mid \exists\, t, u : A^* \,\exists\, o : O \bullet s^0 \overset{t}{\Longrightarrow} \wedge\, t' = tou \wedge \forall\, s : S \bullet (s^0 \overset{t}{\Longrightarrow} s \Rightarrow \neg\, (s \ \overset{o}{\rightarrow}\ )\} \setminus UIA)$$
$$TRC = A^\infty \setminus (UIA \cup UOA)$$

where u is any event sequence including ϵ 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:

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

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

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

Proposition 5.4. Given *PROT*, $SYS(PROT)$, $TM(PROT)$[7], and a relation on trace sets defined as:

$$S \trianglelefteq S' \quad \textit{iff} \quad S \subseteq S' \wedge S = S' \upharpoonright_{I_E \cup I_S},$$

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 $dDIV(TM) = \{t \mid \exists s : s^0 \overset{t}{\Longrightarrow} s \bullet s \in Q\} \cdot \{d^\omega\}$.

Proof:
Follows from Definition 5.6. □

Proposition 5.5. 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, let us prove that $ENV(PROT)$ and $SYS(PROT)$ both conform to *PROT*, i.e. $ENV(PROT)[SYS(PROT)]$ (that is, $TM(PROT) \parallel SYS(PROT)$) is DCIR-free. For divergences, this is obvious since there is no hiding of any events. For chokes, the transition functions of the system and the environment are both based

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

on the same deterministic T; their interaction cannot generate chokes. For illegal stops, a stop state must be in $Q_S \cap Q_E$, but any quiescent hunger state is in $(Q_S \cup Q_E) \setminus (Q_S \cap Q_E)$. So it is impossibe to have illegal stops. For relative starvations, the result is obvious on the environment side since the system does not diverge at all. On the system side, the only possible divergences of the environment are caused by d self-loops, but at that time the environment and the system are both in a Q_E state, on which the system is not hungry, so this causes no relative starvation on the system. Thus, $ENV(PROT)$ and $SYS(PROT)$ conform to $PROT$ as environment and system respectively.

Next, let us prove that $SYS(PROT)$ (with alphabet $A = I_S \cup I_E$) is the least refined conforming one. Without loss of generality, assume a GLAC in normal form, $GLAC \mathrel{\widehat{=}} NC \setminus \Delta$ where $NC \mathrel{\widehat{=}} \|[\overrightarrow{GLAG}]$, 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. Without loss of generality, assume the environment is also in normal form, i.e. $E[\cdot] \mathrel{\widehat{=}} (\|[\overrightarrow{GLAG'}] \parallel [\cdot]) \setminus \Delta'$, and Δ is disjoint from the alphabet A' of $\|[\overrightarrow{GLAG'}] \parallel [\cdot]$. Therefore, $\|[\overrightarrow{GLAG'}] \parallel (NC \setminus \Delta) \setminus \Delta'$ is the same as $CC \setminus (\Delta \cup \Delta')$ where $CC \mathrel{\widehat{=}} \|[\overrightarrow{GLAG'}] \parallel NC$.

In case $E[GLAC]$ is not divergence-free, assume that a trace $\bar{t}$ of CC causes divergence. Then $\bar{t} \upharpoonright_A$ cannot be in $UIA(SYS)$ (since it implies $E[SYS]$ will have a choke on a prefix of $\bar{t} \upharpoonright_A$), and neither in $UOA(SYS) \subseteq UIA(TM)$ (since it implies $ENV(PROT)[GLAC]$ will have a choke). $\bar{t} \upharpoonright_A$ should be in $TRC(SYS)$. But if it is so, and $\bar{t} \upharpoonright_{A'}$ is infinite, $\bar{t} \upharpoonright_{A'}$ will cause divergences in $E[SYS]$. Thus, $\bar{t} \upharpoonright_{A'}$ (and $\bar{t} \upharpoonright_A$) should be finite. However, this implies that $ENV[GLAC]$ has divergences. Contradiction. Hence, $E[GLAC]$ must be divergence-free.

In case $E[GLAC]$ is not choke-free, assume t is a minimal trace of CC causing chokes. As above, $t \upharpoonright_A$ cannot be in $UIA[SYS]$ (due to choke-freedom of $E[SYS]$), nor in $UOA(SYS)$ (due to choke-freedom of $ENV[GLAC]$). It should be in $TRC(SYS)$ ($\subseteq TRC(TM)$). But if it is so, and the choke is in $\overrightarrow{GLAG'}$, this will imply there is a similar choke in $E[SYS]$. If, otherwise, it is on $\overrightarrow{GLAG}$, this will imply $ENV[GLAC]$ will have a similar choke. Contradiction. Thus $E[GLAC]$ must be choke-free.

In case $E[GLAC]$ is not free of illegal stops, assume t is one of its illegal stop traces of CC. Then $t \upharpoonright_{A\cup\Delta}$ must be in $TRC(NC)$ (from the fact that $E[GLAC]$ is choke-free) and $t \upharpoonright_A$ in $TRC(SYS)$ (due to choke-freedom of $E[SYS]$ and $ENV[GLAC]$). Furthermore, $t \upharpoonright_{A\cup\Delta}$ should be in $EIN(NC) \cup PT(NC)$ (due to the definition of illegal stops). If it is in $PT(GLAC)$, then $t \upharpoonright_A$ should be in $EON(SYS)$ (otherwise $E[SYS]$ will have the same illegal stop). But $t \upharpoonright_A$ cannot actually be in $EON(SYS)$ ($\subseteq EIN(TM)$) since that implies $ENV[GLAC]$ is not free of illegal stops. Therefore, $t \upharpoonright_{A\cup\Delta}$ must be in $EIN(NC)$. However, at this time $t \upharpoonright_A$ cannot be in $EON(SYS) \cup PT(SYS)$ ($\subseteq (EIN(TM) \cup PT(TM))$), since that implies $ENV[GLAC]$ is not free of illegal stops. Then the only possible choice is both $t \upharpoonright_{A\cup\Delta}$ in $EIN(NC)$ and $t \upharpoonright_A$ in $EON(SYS)$ ($\subseteq EIN(TM)$). But this is also impossible, since it implies $ENV[GLAC]$ is not free of illegal stops. Contradiction. Thus $E[GLAC]$ must be free of illegal stops.

Finally, 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, the $\overrightarrow{GLAG}$ 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 $\overrightarrow{GLAG}$). Contradiction. Hence, we can conclude that $SYS(PROT)$ is the least refined system conforming to $PROT$.

The least refinedness of $ENV(PROT)$ follows by duality. We omit the details. □

Theorem 5.1. Assume a vector of protocols $\overrightarrow{PROT}$, a generalised context $GC[\vec{x}]$, and a protocol $PROT'$, then $ENV(PROT')[GC[SYS(\overrightarrow{PROT})]]$ is free of DCIRs implies $\forall \overrightarrow{GLAC} \bullet (\overrightarrow{GLAC}$ conforms to $\overrightarrow{PROT} \Rightarrow GC[\overrightarrow{GLAC}]$ conforms to $PROT')$, and vice versa.

Proof:
Follows from Proposition 5.5. □

The theorem tells us that, in order to verify a system consisting of a vector of components, i.e. $GC[\overrightarrow{GLAC}]$, it suffices to verify the system with these components replaced by their protocols. Since a protocol is usually much smaller than its component, the theorem ensures significant leverage in dealing with the state-space explosion problem.

6. An example

Let us give an example to illustrate how we use schedulers and protocol conformance to verify real-world circuits. The system (i.e. $GLAC_{FV}$) is the gate-level circuit in Figure 1 that implements the *FalseVariable* component in Balsa [6]. We verify that the circuit does not conform to the protocol (i.e. $PROT_{FV}$) in Figure 2(a).[8]

We only need to consider safeness here since we will show that $ENV(PROT_{FV})[GLAC_{FV}]$ is not choke-free. Therefore, GLACs and LACs can be used interchangeably below.

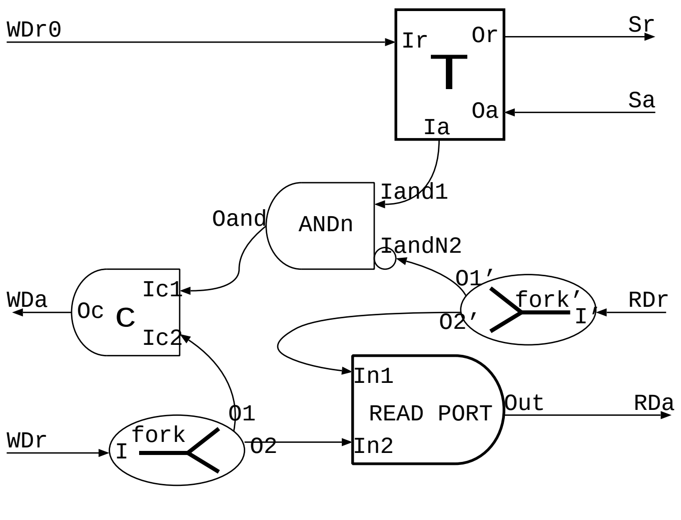


Figure 1. FalseVariable implementation ($GLAC_{FV}$)

[8]The circuit and the protocols in Figure 1 and 2 are provided to us by Luis Plana, Doug Edwards and Andrew Bardsley of Manchester University.

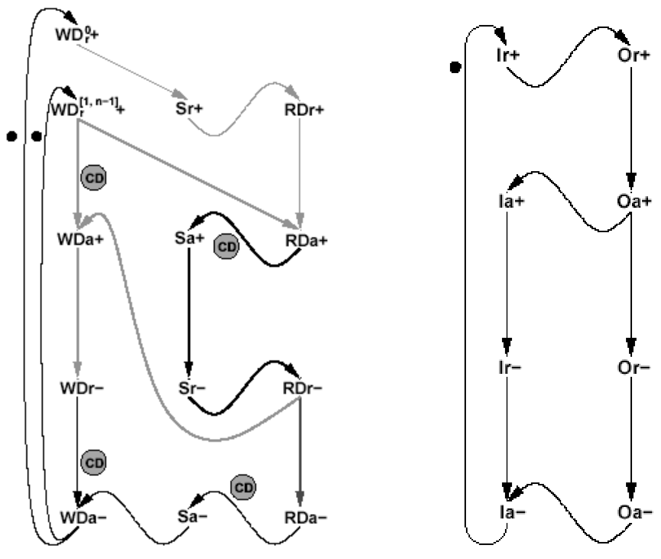
The system $GLAC_{FV}$ is the parallel composition of 6 components. Besides classic gate elements such as *AND*, Muller-C, fork, etc., the circuit also contains a special T component and a *READPORT* component (i.e. $LAC(T)$ and $LAC(READPORT)$). We do not care about the internal implementation of the T component or the *READPORT* component. The circuit is obtained as the parallel composition shown below:

$$GLAC_{FV} = LAG(C) \parallel LAG(fork) \parallel LAC(READPORT) \parallel LAG(fork') \parallel LAG(ANDn) \parallel LAC(T).$$

The protocol of the system ($PROT_{FV}$) is specified using the STG (Signal Transition Graph) [2, 31] in Fig 2(a). Translating the STG in Fig 2(a) to CSP, we obtain the process:

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

Note that the + and - symbols in Figure 2 denote respectively the up and down signal transition on the wires. In the CSP translation, *up* and *down* are used instead. $|||$ is an interleaving operator, while ; is the sequential composition operator[9]. Sequential composition binds stronger than choices, while choices are stronger than parallel composition.



(a) *(b)*

Figure 2. The protocols of FV and T components ($PROT_{FV}$ and $PROT_T$)

[9]To be consistent with theoretical CSP [19], $a;\ a'$ should be written as $a \rightarrow Skip;\ a' \rightarrow Skip$. However, for some CSP variants in asynchronous circuit community, *Skip* can be omitted. For this example, we follow the latter approach.

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 translations are:

$$\begin{aligned}&protocolT = Ir.up;\ Or.up;\ Oa.up;\\&(Or.down;\ Oa.down \mid\mid\mid Ia.up;\ Ir.down);\ Ia.down;\ protocolT\end{aligned}$$

$$\begin{aligned}readport &= (In1.up \mid\mid\mid In2.up);\ Out.up;\ readport'\\readport' &= (\ In1.down;\ (Out.down \mid\mid\mid In2.down)\\&\quad\ \square\ In2.down;\ (Out.down \mid\mid\mid In1.down)\ )\\&\quad\ ;\ readport\end{aligned}$$

For classic gate elements, 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 electronic details. Sometimes, it is much easier to directly give the protocols, and our intuition will suffice to justify them.

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

$$\begin{aligned}fork0 &= I.up;\ (O1.up \mid\mid\mid O2.up);\ fork1\\fork1 &= I.down;\ (O1.down \mid\mid\mid O2.down);\ fork0\end{aligned}$$

The Muller-C element has the protocol:

$$\begin{aligned}protocolC &= (Ic1.up \mid\mid\mid Ic2.up);\ O.up;\ protocolC'\\protocolC' &= (Ic1.down \mid\mid\mid Ic2.down);\ O.down;\ protocolC\end{aligned}$$

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

$$\begin{aligned}andN &= IandN2.up;\ (Iand1.up \mid\mid\mid IandN2.down);\ Oand.up;\ andN'\\andN' &= Iand1.down;\ Oand.down;\ andN\end{aligned}$$

For all these elements, our intuition can easily justify that the components ($LAG(X)$) conform to their protocols ($PROT(X)$). Therefore, to verify that $GLAC_{FV}$ conforms to $PROT_{FV}$, i.e. choke-freedom of $ENV(PROT_{FV})[GLAC_{FV}]$, we only need to show, using Theorem 5.1, that

$$\begin{aligned}&INSTRM = SYS(PROT(C)) \parallel SYS(PROT(fork)) \parallel\\&SYS(PROT(READPORT)) \parallel SYS(PROT(fork')) \parallel\\&SYS(PROT(ANDn)) \parallel SYS(PROT(T)) \parallel TM(PROT_{FV})\end{aligned}$$

is free of chokes.

TM and *SYS* functions above essentially transform a protocol to a CSP process with an isomorphic LTS. 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:

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

The specification of the scheduler is as below:

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

$$\begin{aligned} &scheduler(CN) =\\ &\quad \Box\, x : \mathrm{dom}\, CN \bullet x?z \rightarrow CN(x)!z \rightarrow scheduler(CN)\end{aligned}$$

where CN is the function encoding wire connections.

Next, we supply the process *test_system* to FDR2 [7], where checking deadlock freedom is fully automatic, and indeed discover deadlocks. The deadlocking trace, and consequently the choke trace, is:

$$\begin{aligned} &(WDr.up, I.up)\ (O2.up, In2.up)\ (WDr0.up, Ir.up)\\ &(Or.up, Sr.up)\ (RDr.up, I'.up)\ (O2'.up, In1.up)\\ &(Out.up, RDa.up)\ (Sa.up, Oa.up)\ Ia.up\end{aligned}$$

Note that a choke trace is a sequence of (e^o, e^i) pairs ending with some e^o. The final e^o is the signal transition that has been outputted but caused choke on the receiving component (i.e. the *ANDn* gate here). Therefore, according to Definition 5.3, the circuit in Figure 1 does not conform to protocol of Figure 2(a).

This example only illustrates safeness. More case studies are needed to analyse progress conditions. However, it should be noted that the construction of schedulers can be automated, and hence the circuit verification has the potential to be fully automatic. To handle relative starvation automatically, the FDR2 model checker will need to be modified to capture all the divergences in a process, which was claimed to be no problem by Roscoe [20].

7. Formal relation to other theories

We have obtained a compositional verification theory for asynchronous circuits based on protocol conformance, general enough to capture safeness and progress conditions. As we will demonstrate next, our theory coincides with and extends three important models for asynchronous circuits.

7.1. Trace Theory

Trace theory is based on modelling components/circuits as *Prefixed Closed Trace Structures*,

$$PCTS \mathrel{\widehat{=}} (I, O, S, F)$$

where I and O are the finite input alphabet and output alphabet respectively ($A \mathrel{\widehat{=}} I \cup O$), and S and F are the set of success traces and the set of failure traces (they are regular subsets of A^*). Their union $T \mathrel{\widehat{=}} S \cup F$ is the set of all traces. S is prefix-closed. T must be non-empty, receptive and prefix-closed. There are three operations on PCTSs: parallel composition, hiding and renaming. A PCTS with S empty is called *degenerated*. It corresponds to an LTS with a single *Choke* state. A PCTS with F empty is called failure-free. A PCTS is complete iff its input alphabet is empty.

Based on PCTSs, trace theory developed a verification theory using CPCTS (Canonical PCTS) described below.

Definition 7.1. ([5])
A CPCTS is a PCTS with $F = ((S \cdot I \cup \{\epsilon\}) \setminus S) \cdot A^*$. It can be abbreviated as (I, O, S).

Thus, CPCTS is extension-closed (i.e. chaos after a failure) and output-curtailment closed (i.e. failure caused by input) on F, and S and F are disjoint. One operation M (*mirror*) is defined on non-degenerated CPCTSs. It simply swaps the input and output alphabets, as follows:

$$M((I, O, S)) \mathrel{\widehat{=}} (O, I, S)$$

Proposition 7.1. ([5])
Given a complete CPCTS, it is either failure-free or degenerated.

Relative to the implementation of choke-free systems, trace theory demonstrated that each PCTS has a $\preceq$-equivalent CPCTS substitute.

Definition 7.2. ([5])
$PCTS_i \preceq PCTS_s$ iff $PCTS_i$ is a *safe substitute* (i.e. without introducing more failures) of $PCTS_s$ in any context. $PCTS_i$ and $PCTS_s$ are $\preceq$-equivalent iff $PCTS_i \preceq PCTS_s$ and $PCTS_s \preceq PCTS_i$.

The CPCTS substitutes are calculated using *AM* (Auto-failure Manifestation) and *FE* (Failure Exclusion) transformations on PCTSs. An auto-failure of a PCTS is a failure trace ending with an output event. Given a PCTS, *AM* will make its F output-curtailment closed; *FE* will make its S and F disjoint.

Proposition 7.2. ([5])
Given $AMFE(PCTS) \mathrel{\widehat{=}} FE(AM(PCTS))$, $AMFE(PCTS)$ is a CPCTS and $PCTS$ and $AMFE(PCTS)$ are $\preceq$-equivalent.

Using CPCTSs modelling specifications and PCTSs modelling implementations, their refinement can be checked using the following result.

Theorem 7.1. ([5])
Given $PCTS$ and non-degenerated $CPCTS$, $PCTS \preceq CPCTS$ iff $M(CPCTS) \parallel PCTS$ is failure-free.

Trace theory is closely related to our protocol conformance theory. If only chokes are considered, the two can be shown equivalent. First, a PCTS can be extracted from each LAC as follows:

$$\begin{aligned}
&PCTS(LAC) \mathrel{\widehat{=}} (I, O, S, F)\\
&S = L(LAC, S)\\
&F = (L(LAC, \{Choke\}) \cup (S \cdot I \setminus S)) \cdot A^*
\end{aligned}$$

Recall that a LAC is a finite-state LTS. Augmented with a set of states denoting the accepting states, the LAC will become a finite state automaton. Thus, $L(LAC, S)$ is the language of the automaton with the set of non-choke states S as accepting. It gives us the set of success traces. $L(LAC, \{Choke\})$ is the language of the automaton with the choke states as accepting. It gives us the set of failure traces. It is obvious that $PCTS(LAC)$ has prefix-closed success-trace set (due to the fact that *Choke* is a sink state). Their union is non-empty, receptive and prefix-closed. The complication of the failure trace set is largely due to receptiveness requirements of trace theory.

Lemma 7.1. $PCTS(LAC)$ is not degenerated.

Theorem 7.2. $PCTS(LAC \parallel LAC') = PCTS(LAC) \parallel PCTS(LAC')$.

Theorem 7.3. $PCTS(LAC \setminus C) = PCTS(LAC) \setminus C$.

Lemma 7.2. $PCTS(DT(LAG)) = FE(PCTS(LAG))$, $PCTS(LAG) = AM(PCTS(LAG))$, and for a transparent LAG, $PCTS(LAG)$ is a CPCTS.

Although there is some duality between environments and systems, they are different types of entities in our theory and relative starvations and divergences are not symmetrical on them. Thus, we cannot use a mirror operator in our verification. However, if only chokes and illegal stops are considered, the symmetry and completeness can be recovered, and we can formulate a reduced verification theory using a mirror operator [10].

Definition 7.3. The mirror operation is defined on transparent GLAGs by:

$$M(GLAG) = ((I', O', S', T', s'^0), Q', QH')$$

- $I' = O$, $O' = I$, $S' = S$, $s'^0 = s^0$, $T' = T$,
- $Q' = S \setminus QH$ and $QH' = S \setminus Q$.

Proposition 7.3. M is closed on the transparent LAGs and $LAG = M(M(LAG))$.

Lemma 7.3. Given a transparent LAG, $PCTS(M(LAG)) = M(PCTS(LAG))$.

Similarly, with only chokes, $\preceq_S$ and $\preceq_E$ collapse to a single preorder $\preceq$, and we have a similar verification theorem:

Theorem 7.4. Given LAC and a transparent LAG, $LAC \preceq LAG$ iff $M(LAG) \parallel LAC$ is free of chokes and illegal stops.

In trace theory, efforts have been made to extend the verification to progress conditions. The approach is based on so called *complete trace structures* (CTS), which are not prefix-closed. However, due to the reliance on nondeterministic Buchi automata and infinite games, the method does not possess the intuitive simplicity of the XDI model and induces high complexity in verification, e.g. even for checking the healthiness of a specification (i.e. receptiveness).

[10]Note that both M and DT are defined only on GLAGs, but the induced version on LAGs is self-evident.

7.2. Receptive Process Theory and the XDI model

Receptive process theory (RPT) models components/circuits as:

$$((I, O, S, F), Q)$$

where I and O are the finite input alphabet and output alphabet respectively ($I \cup O = A$), and S and F are the set of safe traces and the set of divergence traces[11]. S is prefix-closed, and F is extension-closed and output-curtailment closed. They are disjoint. Their union $T \mathrel{\widehat{=}} S \cup F$ is the set of all traces. T must be non-empty, receptive and prefix-closed. Thus, (I, O, S, F) constitutes a CPCTS.

Finally, Q is the set of quiescence traces. It is a subset of S and satisfies an *extensibility property*, namely, that every safe trace is output extendable (by zero or more output events) to a quiescence trace and there is no infinitely output extendable safe trace (*ioe*). Hence, S can be derived from Q (that is, due to output extendability, S is the prefix-closure of Q); a trace is a divergence trace iff it is an extension-closed trace (due to the non-existence of *ioe*). Therefore, RPT can be abbreviated as $< I, O, F \cup Q >$.

Two operations are defined on RPT. Hiding (i.e. on outputs) is defined as usual. It will not introduce divergences due to *ioe*. Parallel composition is the usual parallel operator with one exception, namely that parallel will introduce *ioe* and therefore those have to be treated differently, i.e. as divergence traces. Note that in RPT divergences are like chokes in trace theory: some kind of autofailure manifestation transformation needs to be performed on them.

The special treatment of *ioe* is actually based on an interesting assumption on systems and environments. According to their ability of infinite output, systems can be classified as either proactive or reactive. A proactive system is one that is self-motivated and may infinitely output without any input. A reactive system is one that is motivated by the environment, and will stop output (eventually) if the environment does not continue to input. A subclass of reactive systems is called passive if it can absorb infinite input without output, e.g. an observer.

In RPT, all systems are assumed to be reactive systems, and proactive systems are treated as a class of bad reactive systems. If systems only model components, this seems reasonable. But one important weakness (besides the inability to model an oscillator) is that RPT will not be able to model non-terminating complete (or closed) systems. In our theory, proactive systems are the normal cases. The only requirement is that their proactivity cannot be hidden since it may introduce divergences.

Let *IOE* be the set of *ioe* states (see Definition 2.1) of a GLAC. Define:

$$\begin{aligned} &RPT(GLAC) \mathrel{\widehat{=}} (CPCTS, L(GLAC, Q) \setminus F) \\ &CPCTS = AMFE(I, O, L(GLAC, S), L(GLAC, \{Choke\} \cup IOE) \cup (S \cdot I \setminus S)). \end{aligned}$$

Lemma 7.4. $RPT(GLAG) = RPT(DT(GLAG))$.

Theorem 7.5. $RPT(GLAC \parallel GLAC') = RPT(GLAC) \parallel RPT(GLAC')$.

Theorem 7.6. $RPT(GLAC \setminus C) = RPT(GLAC) \setminus C$.

[11] In this paper, we assume the regularity of S and F in RPT to facilitate a comparison with trace theory and our theory. Note also that we use different symbols to identify these sets than in the original theory.

Verhoeff [25, 24] proposed another model, called the XDI model. Each XDI specification ($SPEC \;\widehat{=}\; < I, O, f >$) corresponds to a protocol in our paper ($PROT \;\widehat{=}\; ((I_E, I_S), LAG^D, (Q_E, Q_S))$) such that $I = I_E$, $O = I_S$, and

$$f(t) = \begin{cases} \top & t \in UOA(SYS) \\ \nabla & t \in EON(SYS) \\ \Box & t \in PT(SYS) \\ \triangle & t \in EIN(SYS) \\ \bot & t \in UIA(SYS) \end{cases}$$

where $t \in (I \cup O)^*$. It is not difficult to see that healthiness conditions 1-7 of [24] are satisfied. The reflection operator can also be shown to correspond to the mirror operator in Definition 7.3. Later, Verhoeff and Mallon [14] extended the model to the X^2DI model. X^2DI removes the healthiness conditions 6 and 7. A X^2DI specification roughly corresponds to a transparent GLAG without the well-formedness condition on quiescences.

Both XDI and RPT theories do not handle relative starvations (largely due to the fact that they are finite trace models). However, they both investigate ‘delay insensitive’ (in the sense of having a special event reordering rule) circuits, which are not covered in this paper.

Process spaces [17, 18] are another interesting theory that uses the duality between the environment and system to perform verification. It tackles both safeness and progress in an abstract uniform framework and the parallel operator is abstracted as a conjunction. However, one weakness is that, in order to interpret process spaces as a theory of asynchronous circuits, we need to add healthiness conditions (e.g. receptiveness). It is not always obvious what the healthiness conditions should be, especially when several correctness conditions are combined.

8. Conclusion and future work

We have proposed a compositional verification theory for asynchronous concurrent systems, which can be integrated 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, called $\mathcal{SBDF}$, we give a formal translation from these systems to the process algebra CSP, and show the semantic correspondence (up to isomorphism).
- We reduce asynchronous circuit verification problems to CSP refinement checks, and prove their correctness. This allows us to exploit the power of the process algebra model-checking tools, e.g. FDR2.

From a practical point of view, the protocol/scheduler approach is natural to use, and the extra states introduced by the scheduler are negligible in verification. The main cause of state space explosion in asynchronous circuit verification is unnecessary interleaving. With FDR2, we have shown scalability in verifying certain classical asynchronous circuit problems. For instance, using *chase* compression (a reduction method inspired by partial order semantics), the verification time of the tree arbiter example becomes linear in the size of the tree and we achieve automatic verification of tree arbiters up to 2^{10} ways [27]. Recently, based on analysing concurrency and composition structure of processes, we were able to formulate advanced state space reduction techniques for FDR2 which were inspired by asynchronous circuits verification studied here [29].

However, due to the interleaving nature of the CSP/FDR approach, some aspects of concurrency/causality information are not readily available and fully utilised by our reduction. To make the reduction more effective, a true concurrency approach to CSP is needed, e.g. based on [22].

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, Andrew Bardsley and Luis Plana, 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] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev. Logic Synthesis of Asynchronous Controllers and Interfaces. Springer-Verlag, 2002.

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

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

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

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

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

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

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

[11] 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.

[12] V. Khomenko and M. Koutny. Towards an Efficient Algorithm for Unfolding Petri Nets. CONCUR'01, LNCS 2154, pp. 366-380, Springer, 2001.

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

[14] W. C. Mallon, J. T. Udding and T. Verhoeff. Analysis and Applications of the XDI model. ASYNC 1999: 231-242

[15] Antoni W. Mazurkiewicz. Trace Theory. Advances in Petri Nets, LNCS 255, 1986.

[16] K. L. McMillan. Trace Theoretic Verification of Asynchronous Circuits Using Unfoldings. CAV 1995: 180-195.

[17] R. Negulescu. Process spaces. CONCUR 2000, Springer, 2000.

[18] R. Negulescu. Process Spaces and Formal Verification of Asynchronous Circuits. PhD Thesis, University of Waterloo, Canada, 1998.

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

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

[21] A. Semenov. Verification and Synthesis of Asynchronous Control Circuits using Petri Net Unfoldings. PhD Thesis, University of Newcastle upon Tyne, July 1997.

[22] M.W. Shields. Semantics of Parallelism: Noninterleaving Representation of Behaviour. Springer, 1997.

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

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

[25] T. Verhoeff. A Theory of Delay-Insensitive Systems. Dissertation, Eindhoven University of Technology, May 1994.

[26] X. Wang, M. Kwiatkowska, G. Theodoropoulos and Q. Zhang. Towards a unifying CSP approach for hierarchical verification of asynchronous hardware. ENTCS 128(6), AVOCS 2004, Co-located with CONCUR 2004, UK, 2004.

[27] X. Wang, M. Kwiatkowska, G. Theodoropoulos and Q. Zhang. Opportunities and challenges in process-algebraic verification of asynchronous circuit designs. ENTCS 146(2), FMGALS'05, July, 2005.

[28] X. Wang, M. Kwiatkowska. On process-algebraic verification of asynchronous circuits. Proceedings of ACSD 2006, IEEE Press, Turku, Finland.

[29] X. Wang and M. Kwiatkowska. Compositional state space reduction using untangled actions. To appear in ENTCS, EXPRESS 2006.

[30] Mogens Nielsen, Gordon D. Plotkin and Glynn Winskel. Petri Nets, Event Structures and Domains. Part I. Theor. Comput. Sci. 13(1), 1981.

[31] 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.

A. CSP operators and models

The set of CSP operators used in this paper is:

Prefix operator: $e \rightarrow P$
Sequential composition: $P;\ Q$
Internal choice: $P \sqcap Q$

External choice: $P \,\Box\, Q$
Hiding: $P \setminus \Delta$
Interleaving: $P \,|||\, Q$
Interface parallel: $P \,[\![\, \Delta \,]\!]\, Q$
Renaming: $P[R]$
Recursion: $\mu\ l.F(l)$
Replicated internal choice: $\sqcap\, x : S \bullet P$
Replicated external choice: $\Box\, x : S \bullet P$
Replicated interleaving: $|||\, x : S \bullet P$

In classical CSP [19], stable failures and failure/divergences are the major semantic models used. They are both finite trace models. However, there is a newly developed infinite trace CSP model [20], the $\mathcal{SBDF}$ model, which preserves all the divergence traces in CSP processes.

Given an LTS, the set of finite traces FT is $\{t \mid s^0 \overset{t}{\Longrightarrow}\}$. The set of infinite traces IT is $\{\bar{t} \mid s^0 \overset{\bar{t}}{\Longrightarrow}\}$. The set of divergence traces D is $\{t \mid \exists s \bullet \mathit{divergent}(s) \wedge s^0 \overset{t}{\Longrightarrow} s\}$. The set of stable failures F is $\{(t, \Delta) \mid \exists s \bullet \mathit{stable}(s) \wedge s^0 \overset{t}{\Longrightarrow} s \wedge \forall e \in \Delta \bullet \neg\ (s \overset{e}{\rightarrow}\ )\}$. Given a set of finite sequences, $\mathit{lmt}()$ outputs a set of infinite sequences, each being the limit of a chain of increasing (wrt prefix order) finite sequences belonging to the set. Define $I \mathrel{\widehat{=}} IT \cup \mathit{lmt}(D)$.

Definition A.1. (SBD, SBDF)
$\mathit{SBD}(\mathit{LTS}) = (FT, I, D)$ and $\mathit{SBDF}(\mathit{LTS}) = (F, I, D)$.

The semantics of some simple processes are below:

$\mathit{SBD}(\mu\, l.(e \rightarrow l)) = (\{e\}^*, \{e^\omega\}, \{\})$
$\mathit{SBD}(\sqcap\, i : \mathbb{N} \bullet P(i)) = (\{e\}^*, \{\}, \{\})$, where $P(0) = \mathit{Stop}$ and $P(n) = e \rightarrow P(n-1)$
$\mathit{SBDF}(\mu\, l.(l) \sqcap \mathit{Stop}) = (\{(\epsilon, \Delta) \mid \Delta \subseteq A\}, \{\}, \{\epsilon\})$
$\mathit{SBDF}(\mu\, l.((a \rightarrow l) \,\Box\, (b \rightarrow \mathit{Stop}))) =$
$\quad (\{(t, \Delta) \mid t \in \{a\}^*\{b\} \wedge \Delta \subseteq A\} \cup \{(t, \Delta) \mid t \in \{a\}^* \wedge \Delta \subseteq (A \setminus \{a, b\})\}, \{a^\omega\}, \{\})$