*Ji He and Kenneth J. Turner. Verifying and Testing Asynchronous Circuits using LOTOS (extended version). In Tommaso Bolognesi and Diego Latella, editors, Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XIII/PSTV XX), Kluwer Academic Publishers, London, UK, October 2000.*

# Verifying and Testing Asynchronous Circuits using LOTOS

**Ji He and Kenneth J. Turner**
University of Stirling, Scotland FK9 4LA
*Email h.ji@reading.ac.uk, kjt@cs.stir.ac.uk*

1st June 2000

**Abstract**

It is shown how DILL (Digital Logic in LOTOS) can be used to specify, verify and test asynchronous hardware designs. Asynchronous (unclocked) circuits are a topic of active research in the hardware community. It is illustrated how DILL can address some of the key challenges. New relations for (strong) conformance are defined for assessing a circuit implementation against its specification. An algorithm is also presented for generating and applying implementation tests based on a specification. Tools have been developed for automated verification of conformance and generation of tests. The approach is illustrated with three case studies that explore speed independence, delay sensitivity and testing of sample asynchronous circuit designs.

**Keywords**: Conformance Testing, Hardware Description, Hardware Verification, LOTOS

# 1 Introduction

## 1.1 Background

DILL (Digital Logic in LOTOS, e.g. [9, 10]) is an approach, a language and a toolset for specifying, analysing and testing digital hardware designs. The authors have developed an extensive library of typical hardware components, and methods for dealing with synchronous (clocked) circuits. Although DILL has language facilities for inclusion of standard circuit components, it is really just a veneer on top of LOTOS. DILL can therefore exploit the rich theories and tools for LOTOS. The recent developments described in this paper have extended DILL to asynchronous (unclocked) circuits.

Conventional digital circuits operate under control of a master clock that synchronises the major components. The design of such synchronous circuits is well understood. The clocked nature of designs avoids problems due to variations in speed among the components. However, asynchronous circuit design is increasingly attracting attention. These circuits operate at the speed of individual components, without artificially limiting them to some clock rate. They may thus be capable of faster operation than synchronous circuits. Unfortunately asynchronous circuits are much harder to design and analyse. Indeed there are many design strategies which aim to achieve different goals such as independence from the delays or speed of circuit elements. The challenge is to allow each component to operate as fast as possible while preserving the circuit function.

Verification of asynchronous circuits, especially speed-independent and delay-insensitive ones, is an active research topic (e.g. [2, 4]). Rigorous testing of asynchronous circuits is still in its infancy. This paper presents solutions for verification of asynchronous circuits and for deriving rigorous implementation tests from their specifications. It thus makes a contribution to the evolving understanding of asynchronous circuit design. The paper also shows a novel application of protocol design techniques (LOTOS, conformance testing) to a new domain that is exciting industrial interest.

## 1.2 Related Work

Formal techniques that have been used for asynchronous circuits include CIRCAL [1], CSP and Delay-Independent Algebra. Of these, DILL most closely resembles CIRCAL in that both have a behavioural basis in process algebra, and both have been used in (a)synchronous circuit design. However, the integrated data types in LOTOS makes it much more expressive than CIRCAL. In the authors' experience, LOTOS can be used successfully at a variety of abstraction levels whereas CIRCAL appears to be less effective at higher levels. It is very difficult to use CIRCAL for specifying relatively complicated behaviour.

Like DILL, many other asynchronous verification approaches define relations that judge correctness of a circuit design. The relations *confor* and *strongconfor* defined in this paper resemble those introduced by others, e.g. *conformance* [2], *decomposition* [4] and *strong conformance* [6]. It is not possible to detect deadlocks and livelocks with *conformance* and *decomposition*. Although *strong conformance* can do this, it requires an implementation not to produce less outputs than its specification. This excludes the possibility of applying the relation to non-deterministic specifications. As will be seen, the *(strong)confor* relations defined in this paper have clear advantages over those mentioned above.

For validating hardware designs, test cases are in practice manually defined or are randomly generated. More rigorous approaches use traditional software testing techniques or state machine representations. [13] makes use of higher-level state machine specifications, but cannot handle non-deterministic specifications. In the DILL approach, tests are derived from higher-level specifications in a novel adaptation of the conformance testing theory developed for protocols.

### 1.3 Structure of Paper

Section 2 discusses various kinds of asynchronous circuit components and how they can be specified in DILL. Section 3 then explains how asynchronous circuit implementations can be verified against their specifications, using the notion of conformance relations. The theory and tools also allow automated derivation of tests for asynchronous circuits. Section 4 shows how an asynchronous FIFO (First-In First-Out buffer) can be verified to exhibit liveness and speed independence. Automatically generated tests are also presented. The *Or-And* example in section 5 demonstrates the subtleties in verifying speed independence and delay insensitivity. Two plausible implementations are shown to have differing properties. Finally, section 6 uses the example of a selector to demonstrate how the approach copes with specifications that allow non-deterministic implementations.

## 2 Specifying Asynchronous Circuit Components

### 2.1 Classes of Asynchronous Circuits

Asynchronous circuits exhibit a variety of forms due to the different delay and environment assumptions made. An asynchronous circuit can only behave correctly when these assumptions are met. Some of the better-known design approaches include the following.

**DI (Delay-Insensitive)** circuits are the most robust class in the asynchronous circuit family since they take the most pessimistic view about delays and the environment. Delays in components and wires are assumed to be unbounded (but finite). DI circuits can operate correctly regardless of actual delay magnitudes. Most meaningful DI circuits require more than basic logic gates, so specialised components are defined [11].

**QDI (Quasi Delay-Insensitive)** circuits augment the delay model of delay-insensitive circuits with the use of isochronic forks. These are branching connections on which the difference of delay magnitudes is negligible. This is a good compromise for building practical circuits using single-output gates.

**SI (Speed-Independent)** circuits have gates with unbounded delay, but wires have zero delay. If all gates have just one output, SI and QDI are actually identical.

Specifying bounded delays needs a formalism that supports quantitative timing specification. This paper studies the classes above since they assume unbounded delays and so are suitable for LOTOS. (Quasi) delay-insensitive designs can be easily changed to speed-independent circuits by inserting artificial delay components. Since unbounded delay plus unbounded delay is still unbounded, most of the wire delays can be absorbed into the preceding components. Only forks and components with more than one output need special treatment.

The rest of the paper therefore mainly discusses speed-independent designs. Happily these are a good match to the DILL approach since component delays are unbounded, just like the interval between consecutive LOTOS events. In DILL, component ports are connected by synchronising their LOTOS events. This actually assumes that delay on the connecting wires is negligible, an assumption that is also adopted by speed-independent circuits. Speed independence is closely related to the concept of semi-modularity. If new inputs cannot can change any pending outputs, the design is termed semi-modular. Semi-modularity is commonly used as a correctness criterion for speed independence, since the violation of semi-modularity causes speed-dependent behaviour.

### 2.2 Modelling Speed Independence

In this section, basic logic gates are used to illustrate how to model speed independence (really semi-modularity) of asynchronous circuits. In [9, 10], the specification of basic logic gates allows new inputs to pre-empt pending outputs. Some of these input transitions may change the levels of pending outputs, resulting in the violation of semi-modularity. Consider a *Nand2* gate (two-input 'not and') whose inputs and output *Ip1, Ip2, Op* are initially *1, 1, 0*. After *Ip1* changes to *0*, its output should change to *1*. If *Ip1* changes back to *1* before output happens, the output may either undergo the *1* to *0* transition or remain on *0*. This depends on the speed of the gate. To respect semi-modularity, such inputs have to be forbidden.

```
process Nand2 [Ip1,Ip2,Op] (dtIp1,dtIp2,dtOp:Bit) : noexit :=
  let newOp:Bit = dtIp1 nand dtIp2 in                 (* potential output *)
    (
      Ip1 ?newIp1:Bit [(newIp1 ne dtIp1) and          (* first input changes *)
        ((dtOp eq newOp) or                           (* no new potential output *)
          ((dtOp ne newOp) and                        (* there is potential output *)
          ((newIp1 nand dtIp2) eq newOp)))];          (* but no change is needed *)
      Nand2 [Ip1,Ip2,Op] (newIp1,dtIp2,dtOp)          (* continue behaviour *)
    []
      Ip2 ?newIp2:Bit [(newIp2 ne dtIp2) and          (* second input changes *)
        ((dtOp eq newOp) or                           (* no new potential output *)
          ((dtOp ne newOp) and                        (* there is potential output *)
          ((newIp2 nand newIp1) eq newOp)))];         (* but no change is needed *)
      Nand2 [Ip1,Ip2,Op] (dtIp1,newIp2,dtOp)          (* continue behaviour *)
    []
      Op !newOp [dtOp ne newOp];                      (* new output produced *)
      Nand2 [Ip1,Ip2,Op] (dtIp1,dtIp2,newOp)          (* continue behaviour *)
    )
endproc
```

Note that the specification is partial in that inputs are forbidden at certain points. An input offer can happen only when there is no potential output, or when the new input cannot alter the potential output.

The usual DILL specification of a logic gate [9, 10] is input-receptive. But it possesses inertial delay, i.e. the gate will not react to short input pulses. The model above is stricter than that of [9, 10], and can be used for checking if a circuit is semi-modular or not.

### 2.3 Specifying Asynchronous Circuit Components

Asynchronous circuit design is a specialised discipline that uses special-purpose components in addition to basic logic gates. These special components are treated as fundamental, even though their implementation may derived from simpler elements. The special components are assumed to exhibit properties such as speed independence or delay insensitivity. To give a flavour of the approach and to show that their specifications in DILL are straightforward, a sampling of the components is specified below. For brevity an abbreviated syntax is used for process definitions. It is also common practice [3] to omit signal levels when specifying asynchronous circuits, since the levels strictly alternate. For example the input sequence *Ip !0, Ip !1, Ip !0* can be represented as *Ip, Ip, Ip*. The following outline specifications respect the requirement of semi-modularity.

**Wires** are the simplest components. In high-speed circuits such as asynchronous designs, even connections can contribute delays. They are not needed for speed-independent circuits since delays on wires are assumed to be zero. But when a (quasi) delay-insensitive circuit is transformed to a speed-independent design, delays on wires may have to be explicitly specified.

Wire [Ip,Op] := Ip; Op; Wire [Ip,Op]

**Fork** components are also necessary when transforming a (quasi) delay-insensitive design to speed-independent form. A fork has one input *Ip* and two outputs *Op1, Op2*. The value on input *Ip* is fanned out to *Op1* and *Op2*. Because of delays, the two outputs may occur at different times. New input has to wait until both outputs have been produced.

Fork [Ip,Op1,Op2] := Ip; (Op1; **exit** ||| Op2; **exit**) ≫ Fork [Ip,Op1,Op2]

**C-Elements** are very important in asynchronous design. A C-Element (named after its conventional output *C*) is used as a transition synchroniser since its output can change only after both inputs have changed. For this reason, it is sometimes also called a join element. A C-Element has two inputs *Ip1, Ip2* and an output *Op*. The output changes to *1* when both inputs have changed to *1*, and changes to *0* when both have changed to *0*.

C-Element [Ip1,Ip2,Op] := (Ip1; **exit** ||| Ip2; **exit**) ≫ (Op; C-Element [Ip1,Ip2,Op])

**Merge** components copy input signals *Ip1, Ip2* to a single output *Op*.

Merge [Ip1,Ip2,Op] := Ip1; Op; Merge [Ip1,Ip2,Op] [] Ip2; Op; Merge [Ip1,Ip2,Op]

**Selectors** take a single input *Ip* and non-deterministically output on *Op1* or *Op2*.

Selector [Ip,Op1,Op2] := Ip; (**i**; Op1; **exit** [] **i**; Op2; **exit**)≫ Selector [Ip,Op1,Op2]

**Sequencers** have two data inputs *Ip1, Ip2*, a control input called *C*, and two data outputs *Op1, Op2*. They wait for an *Ip1* or *Ip2* signal plus *C*, and then produce the corresponding output signal.

Sequencer [Ip1,Ip2,C,Op1,Op2] := (S1 [Ip1,Op1] ||| S2 [Ip2,Op2]) |[Op1,Op2]| S3 [C,Op1,Op2]
**where**
S1 [Ip1,Op1] := Ip1; Op1; S1 [Ip1,Op1]
S2 [Ip2,Op2] := Ip2; Op2; S2 [Ip2,Op2]
S3 [C,Op1,Op2] := C; (**i**; Op1; S3 [C,Op1,Op2] [] **i**; Op2; S3 [C,Op1,Op2])

**Latches** are the storage elements in asynchronous circuits. A latch has two data inputs *Ip1, Ip2*, a control input *C*, and two data outputs *Op1, Op2*. A latch waits for exactly one *Ip1* or *Ip2* input and a *C* input, then the corresponding output is produced. In contrast to a sequencer, the environment must guarantee mutual exclusion of inputs.

Latch[Ip1,Ip2,C,Op1,Op2] :=
((Ip1; **exit** ||| C; **exit**) ≫ Op1; Latch [Ip1,Ip2,C,Op1,Op2])
[]
((Ip2; **exit** ||| C; **exit**) ≫ Op2; Latch [Ip1,Ip2,C,Op1,Op2])

**RGD Arbiters** are named after their signals: Request, Grant, Done. They have four inputs *R1, D1, R2, D2* and two outputs *G1, G2*. If the arbiter receives two simultaneous requests, it grants exactly one of them and delays the other. Request $R_i$ is followed by grant $G_i$ and then the done signal $D_i$. The time intervals *G1..D1* and *G2..D2* are guaranteed to be mutually exclusive.

RGD [R1,G1,D1,R2,G2,D2] := (S1 [R1,G1] ||| S2 [R2,G2]) |[G1,G2]| S3 [G1,D1,G2,D2]
**where**
S1 [R1,G1] := R1; G1; S1 [R1,G1]
S2 [R2,G2] := R2; G2; S2 [R2,G2]
S3 [G1,D1,G2,D2] := (**i**; G1; D1; S3 [G1,D1,G2,D2]) [] (**i**; G2; D2; S3 [G1,D1,G2,D2])

### 2.4 Input (Quasi-)Receptiveness

In LOTOS, communication between processes is based on symmetric synchronisation at a gate. Thus an event can happen only when all processes offer events at this gate. If one of the processes is not able to do so, the other processes just wait or participate in another event if possible. However, digital hardware has a clear distinction between inputs and outputs. Signals are absorbed at inputs and are produced at outputs. A hardware component can never refuse an input signal, and the output signals it produces can never be blocked by others.

A specification is said to be input-receptive if every input is allowed in every state. In such a case, the DILL model represents the real circuit faithfully. However input-receptive specifications cannot be written for most asynchronous circuit components since unexpected inputs are not permitted. Analysing such partial specifications has the disadvantage of not being exact since the behaviour of a LOTOS model is only a subset of the behaviour of the real circuit. One way to address this is by explicit deadlock if unexpected inputs arrive.

If more accurate analysis is required, input quasi-receptive specifications should be written. Informally, a DILL specification is input quasi-receptive if it can always participate in all input events except when deadlocked. As an example, the specification of a wire is partial in that input is not allowed when the wire wants to produce its output. An input quasi-receptive specification can be obtained by adding a choice when there is a potential output. A wire might thus be specified as:

```
process Wire [Ip,Op] (dtIp:Bit) : noexit :=
  Ip ?newIp1:Bit [dtIp ne newIp1];               (* new input *)
  (
    Op !newIp1; exit (newIp1)                    (* new output *)
  []
    Ip ?newIp2:Bit [newIp2 ne newIp1];           (* forbidden input causes ... *)
    stop                                         (* forced deadlock *)
  )
>>
  accept newIp:Bit in Wire [Ip,Op] (newIp)       (* continue behaviour *)
endproc
```

It is not straightforward to transform a specification with more than just sequence and choice operators into input quasi-receptive form. In such a case, a partial specification can be used to generate the corresponding LTS (Labelled Transition System). An LTS is actually a LOTOS specification in the form of sequence and choice operators. An input quasi-receptive specification can be obtained by modifying the LTS. For a state that cannot participate in all input events, outgoing edges leading to deadlock are added for missed inputs. This method works very well for LTSs without internal events. But subtle problems can arise for those containing internal events. Suppose state *S* accepts an input *Ip1*, performs an internal action, and ends up in state $S'$ where *Ip2* can be accepted. It would be incorrect to add a transition to state *S* that accepts *Ip2* and leads to deadlock. That is, *S* accepts *Ip2* only through an internal action.

When a specification is considered from the point of view of input receptiveness, internal events become less meaningful. Internal events mean the environment has no effect on choices. In the context of digital design, a circuit produces outputs without influence from its environment. Therefore all outputs should be preceded by internal events. If internal events are omitted, the environment has the opportunity to choose which outputs to accept and which to refuse; this is not a proper model of real circuits. However if the environment is input-receptive, it loses its selective power: it will accept whatever the circuit produces, even though there are no internal events in the circuit specification. For this reason, LTSs with internal events are determinised before outgoing edges are added to create input quasi-receptive specifications. An LTS is input quasi-receptive if, after determinisation, all states except terminal ones can accept all inputs.

# 3 Asynchronous Circuit Verification and Testing

## 3.1 Verifying Asynchronous Circuit Designs

The characteristics of asynchronous circuits have implications for verification. As it is more difficult to specify components in an input (quasi-)receptive manner, verification may still be based on using components that are not input-receptive. The verification may, however, not be exact in that some problems may not be discovered. Input quasi-receptive specifications result in a larger state space and thus make verification more difficult.

A structural implementation (a detailed design) normally has much more behaviour than its abstract specification. This is not unique to DILL and also applies to many other design methods. This makes it unrealistic to use equivalence as the criterion for correct design. Equivalence requires that a specification and its implementation have same behaviour under all possible environments. This requirement would usually be too strong since practical circuits work correctly only in well-behaved environments.

Assumptions about environments have to be made since these are often not given for asynchronous circuits. When an environment is not explicit, many methods simply assume the mirror of a specification as the environment of its implementations [2]. If $S$ is the abstract specification and $I$ is the implementation specification, verification means comparing $S \,||\, I$ with $S$ or checking if a logic formula holds for $S \,||\, I$. But verifying $S \,||\, I$ is not always satisfactory. When an implementation can accept more inputs than its specification does, $S \,||\, I$ restricts the inputs considered to only those in the specification. This assumes that the environment does not provide extra inputs, so inputs that are accepted only by the implementation are ignored when verifying the joint behaviour. This is reasonable, but permits an implementation to produce more outputs than a specification. This is undesirable since an implementation producing unexpected output for legitimate input is normally erroneous. Moreover, when a specification is non-deterministic this method may exclude correct deterministic implementations.

The key point here is the different roles of inputs and outputs in digital circuits. An implementation passively accepts inputs, so only those available from the environment make sense. An implementation actively produces outputs, over which the environment has no influence. A LOTOS specification however does not distinguish inputs and outputs. When a LOTOS specification is used as an environment, it restricts inputs and outputs equally.

When an implementation is specified in an input (quasi-)receptive way, a distinction is made between inputs and outputs. If its environment is also receptive, it will deadlock on unexpected outputs from an implementation. However, it is very hard to extract an input quasi-receptive environment from a behavioural specification – especially if this is complicated or contains internal events. An alternative method is therefore adopted for verifying asynchronous circuits. Relations are defined that take into account the difference between input and output. These relations do not require (quasi-)receptiveness of the environment or the implementation, and are intuitive criteria for correctness of asynchronous circuits.

## 3.2 Input-Output Conformance for Asynchronous Circuits

Although many relations have been defined to characterise the relationship between two LTSs, they are not very helpful for verifying asynchronous circuits. This is especially true when the environment of a circuit is not explicitly supplied. Two new relations, *confor* and *strongconfor*, are therefore defined to assess (strong) conformance of an implementation to its specification. These relations were inspired by *ioconf* and *ioco* in [14] for conformance testing of communications protocols.

Suppose *Spec* is an abstract specification of a circuit and *Impl* is its implementation specification. *Spec* may be partial in the sense that in some states it does not accept some inputs, i.e. it is not input-receptive. An input is absent if the environment of a circuit does not provide it, if the behaviour of the circuit upon receiving the input is not of interest, or if the behaviour is undefined. Although all circuits are potentially able to accept all inputs at any time, most specifications are partial to avoid too much detail. *Impl* may be partial or total in the sense of input receptiveness.

Suppose that *sp* is a state of *Spec* and that *im* is the corresponding state in *Impl*. To define the *confor* relation, first consider the input transitions that *sp* and *im* can engage in. If an input *ip* is acceptable in *sp*, it is reasonable to require that *ip* is also accepted in *im*. On the other hand, if *im* can accept an input which is not acceptable in *sp*, this input and all the behaviour afterwards can be ignored. Since the environment will never provide such an input, or even if it is provided, such behaviour is not of interest. In short, the inputs acceptable in *sp* should be a subset of those acceptable in *im*.

As far as outputs are concerned, if *sp* can produce *op* then a correct implementation should also produce it. If *sp* cannot produce a certain output, neither should its implementation. However when a specification is allowed to be non-deterministic, it is too strong to require *im* to produce exactly the same outputs as *sp* since a deterministic implementation could produce a subset of the outputs. A suitable relation should thus require output inclusion instead of output equality. Unfortunately a circuit that accepts everything but outputs nothing may also be qualified as a correct implementation. The special $\delta$ 'action' overcomes this weakness by indicating the absence of output. Like any other output action, if $\delta$ belongs to the output set of *im* it must be in the output set of *sp* for conformance to hold. In other words, *im* can produce nothing only if *sp* can do nothing as well.

In the above discussion, *sp* and *im* are not actually LTS states but are all possible situations that a circuit may be in after a certain input-output sequence. Because $\delta$ is also involved in the sequence, the state spaces of both specification and implementation are transformed into automata that are explicitly labelled with $\delta$. The input-output sequences are actually traces of the specification automaton. Formally, let implementation $i \in \mathcal{LTS}(L_I \cup L_U)$ and specification $s \in \mathcal{LTS}(L_I \cup L_U)$. Then:

$$\begin{array}{lll} i \text{ confor } s & =_{def} & \forall\sigma \in \text{STrace(s)}: \ \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)\text{and} \\ & & \text{if } i \text{ after}\sigma \neq \emptyset : \text{in}(s\text{after}\sigma) \subseteq \text{in}(i \text{ after } \sigma) \\ i \text{ strongconfor } s & =_{def} & \forall\sigma \in \text{STrace(s)}: \ \text{out}(i \text{ after } \sigma) = \text{out}(s \text{ after } \sigma)\text{and} \\ & & \text{if } i \text{ after}\sigma \neq \emptyset : \text{in}(s\text{after}\sigma) \subseteq \text{in}(i \text{ after } \sigma) \end{array}$$

$L_I$ and $L_U$ refer to the inputs and outputs from the point of view of the implementation. The inputs and outputs of an automaton after some trace are computed by *in* and *out*. The *after* operator yields an automaton after it has executed a given trace. *STrace* generates a suspension trace.

To define suspension traces, the transition relation is extended with the refusal of output actions: self-loop transitions labelled with $L_U$ indicate that no output action can occur. Refusal to output can also be expressed using $\delta$, which is regarded as an output action distinct from $L_I$ and $L_U$. A suspension trace thus contains ordinary actions and $\delta$ actions. If $L_\delta$ denotes $L \cup \delta$, a suspension trace $\sigma \in L_\delta^\star$.

The *confor* relation requires that, after a suspension trace of *s*, the outputs that an implementation *i* can produce are included in what *s* can produce. If *i* can follow the suspension trace, the inputs that *s* can accept are also accepted by *i*. *strongconfor* has a similar definition except that output inclusion is replaced by output equality. Normally *confor* is used when a specification and an implementation are deterministic, while *strongconfor* is used when an implementation is more deterministic than a specification. The *(strong)confor* relations are more easily observed if the LTSs of specifications are transformed to suspension automata.

A suspension automaton $\Gamma_p$ of an LTS *p* is obtained by determinising *p* and adding the necessary $\delta$ transitions. The suspension traces of *p* coincide with the traces of its suspension automaton $\Gamma_p$. In addition, for all $\sigma \in L^\star$, $out(\Gamma_p \; after \; \sigma) = out(p \; after \; \sigma)$; see [14] for the proof. Therefore checking *(strong)confor* can be easily reduced to checking trace inclusion on suspension automata. Only traces without $\delta$ transitions are checked for *confor*, while all the traces of a suspension automaton are checked for *strongconfor*.

A verification tool *VeriConf* has been developed to check the *(strong)confor* relations using the programming interface of CADP (Cæsar Aldébaran Development Package [5]). Briefly, CADP is exploited to generate LTSs of both specification and implementation. Then the verifier is used to produce the suspension automata from the LTSs and to compare the automata according to the relations. The verifier has been successfully used in analysing several asynchronous circuits, including the examples in the following sections.

### 3.3 Testing Asynchronous Circuits

Conformance testing requires several ingredients: a formal specification, an IUT (Implementation Under Test), a test suite, and a relation that checks correctness of the implementation against the specification. There should preferably be a test generation algorithm that produces test suites automatically. An IUT may be a product, a formal specification, or even an informal specification. Presuming that every IUT has a formal model is referred to the test hypothesis. Note that only the *existence* of a model is assumed. In this paper, implementations are modelled as IOLTSs (Input Output Labelled Transition Systems).

An IOLTS *p* is an LTS whose actions $L$ are partitioned into inputs $L_I$ and outputs $L_U$, and whose input actions are always enabled in any state. This class of system is denoted by $\mathcal{IOLTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$. Specifications, however, are still modelled as an LTS to permit an abstract view of behaviour. Such specifications are interpreted as incompletely specified IOLTSs where some inputs are not specified in some states. The intention of incompleteness might be implementation freedom, or because the environment will not provide undesirable inputs.

Several implementation relations have been defined by others between LTSs and IOLTSs. Some of these relations, such as the one analogous to testing preorder, are too strong in that they require *specifications* to be IOLTSs. This is obviously impractical in most cases. The *ioco* relation has been defined to support conformance testing. The *ioco* relation is very similar to *confor*. The difference is that implementations accepted by *confor* may not be input-receptive, whereas *ioco* assumes that implementations are modelled as IOLTSs (and can thus always accept all inputs). Consequently, the input inclusion condition required by *confor* is always satisfied in the case of *ioco*.

A test suite is a set of test cases. For practical reasons, test cases must have finite behaviour. In addition they should be deterministic to allow a tester to have control over test execution. This requires that test cases have no choices between multiple input actions or between input and output actions, as both introduce undesirable non-determinism during a test. As a result a state of a test case is either terminal, offers one input to the implementation, or accepts all possible outputs from the implementation (including the $\delta$ action). The terminal states of a test are labelled with *Pass* or *Fail* to yield a verdict. When an implementation is tested, it will stop only in a pass or fail state. Since an implementation can be non-deterministic, different terminal states can be reached with different test runs of the same test case. Only when an implementation passes all possible test runs is it said to pass the test case.

A test case is modelled as an $\mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$. As before, $\delta$ means a state cannot produce any output. Test cases are obtained by a finite recursive application of the following non-deterministic choices: terminate the test case; give a next input to the implementation; or check the next outputs of the implementation. The first choice terminates the generation procedure to ensure a test stops at some point even though the specification may include infinite behaviour. The second choice will never result in deadlock as inputs are always enabled. The third choice ensures failure if an implementation produces an output not belonging to *out*($\Gamma$). This test generation algorithm guarantees sound test cases with respect to *(strong)confor*, and the set of test cases that can be obtained is complete; see [14] for the proof.

In the DILL approach to testing digital circuit designs, a circuit is specified in LOTOS (whose semantics is given by an LTS). The implementation of the same circuit is described by VHDL (VHSIC Hardware Description Language [8]). The behaviour of a VHDL program is presumed to be modelled by an IOLTS that is merely assumed to exist – it need not be known explicitly. The test suite for a circuit is generated by an algorithm based on that of [14]. The authors have extended CADP to generate hardware test suites automatically from the suspension automaton of the specification. A VHDL testbench executes and evaluates the test cases. If there is an inconsistency between the formal specification and its VHDL

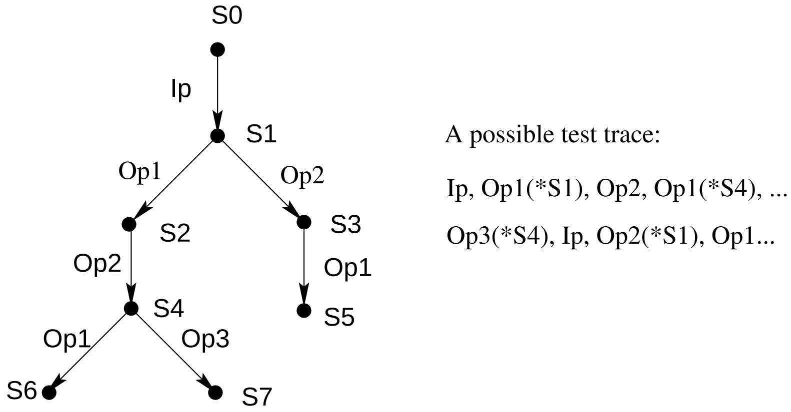


Figure 1: Test Trace for Nodes with Several Outputs

implementation, the implementation is regarded as incorrect.

The test cases generated in DILL approach have the form of traces rather than trees. This allows easy measurement of test coverage, and automatic execution of test cases. A test suite cannot usually cover the entire behaviour of a specification as this is normally infinite. The strategy is therefore to cover all transitions in a transition tour that addresses the Chinese Postman problem. As suspension automata may not be strongly connected, it is not possible to make direct use of conventional transition tour algorithms. Instead the approach of [7] is used because it is suitable for all kinds of directed graphs. Depth-first search is used until an unvisited edge cannot be reached. Breadth-first search is then employed to find an unvisited edge, and then depth-first search recommences. The authors have developed the *TestGen* tool that realises this algorithm using the CADP application programming interface.

The problems caused by non-determinism can be solved by marking contradictory output branches in a suspension automata. This situation arises if an output may not be matched by the implementation under test since other outputs are permitted. This method is not so effective when the behaviour of an implementation is non-deterministic. The problem is that when an inconclusive verdict is reached, a test run is aborted and other test cases are applied. However, the test case could be still useful if other neighbouring outputs can be found so that the test run may continue.

Contradictory branches in the suspension automaton are therefore marked with '⋆' and the corresponding state label. Obviously, outputs with the same marks in a test suite are neighbours in the corresponding suspension automaton. In this way the branching structure of the automaton tree is reflected in a test case. The transition tour algorithm is able to cover all the transitions in a suspension automaton. If an implementation differs from all the outputs with a certain mark, a fail verdict should be recorded. This technique requires a testbench that is able to search the whole test suite for marks.

Figure 1 is an example of this technique. If an implementation has the behaviour *Ip, Op1, Op2, Op3, ...* it will follow *Ip, Op1, Op2* in a test run. But when output *Op3* fails at *Op1(⋆S4)*, the testbench must look for another output with the same mark to see if *Op3* can be matched. In this case it can find *Op3(⋆S4)* and continue testing. If an implementation behaves as *Ip, Op3, ...* then there will be no output marked with *(⋆S1)* that can match *Op3*; the implementation would be regarded as incorrect.

The testbench would normally search the remainder of a test trace when an inconclusive point is met so that testing can go forward. However such marks sometimes exist only in the previous part of the trace, forcing the search to go backward. This means that loops may arise during testing, so the testbench needs a strategy to avoid this.

The testbench also needs to maintain a timer. Real components do not have unbounded delays, so when a $\delta$ transition is encountered the testbench must record the time that elapses. If there is no output within a certain period, the $\delta$ transition can be regarded as having occurred; otherwise a failure verdict must be given. The value of the timer reflects the delays in the real circuit. A testbench will also have to decide when to provide inputs. For test case *T1* in the later example of figure 3, if *InF !0* is provided too late after the first input *InF !1* then an output may have already been produced. The behaviour should be fully explored by other test cases such as *T2*. But the tester might not be aware of the interrelationships among tests. *T1* might therefore be used at the risk of producing faulty test results.

## 4 Case Study: An Asynchronous FIFO

As a typical circuit, an asynchronous FIFO (First In First Out buffer) is specified and analysed. The FIFO has two inputs *InT, InF* and two outputs *OutT, OutF*. Its inputs and outputs use dual-rail encoding in which one bit needs two signal lines. The pair of *T/F* (true/false) signal values *1/0* corresponds to data value *1*, while the pair *0/1* corresponds to *0*. A signal of *0* on both lines indicates idle, which means there is no valid data. Lines have to be reset to idle between two transmissions. Suppose a FIFO with one stage is initially empty. It can accept either *1* or *0* on receipt of *InT* or *InF*. The data is delivered to the output lines. After one successful transmission, the input and output lines that have been raised return to *0* to wait for other data. The behaviour of one stage can be easily specified:

```
process Stage [InT,InF,OutT,OutF] :noexit :=          (* one FIFO stage *)
  InT !1; OutT !1; InT !0; OutT !0;                   (* input/output 1 then idle *)
  Stage [InT,InF,OutT,OutF]                           (* continue behaviour *)
[]
  InF !1; OutF !1; InF !0; OutF !0;                   (* input/output 0 then idle *)
  Stage [InT,InF,OutT,OutF]                           (* continue behaviour*)
endproc
```

A FIFO with several stages can be obtained by composing instances of this process. For example, a FIFO with two stages and internal signals *IntT*, *IntF* is specified as:

```
process Spec [InT,InF,OutT,OutF]                      (* FIFO specification *)
  hide IntT,IntF in                                   (* internal signals *)
    Stage [InT,Inf,IntT,IntF]                         (* first stage *)
 |[IntT,IntF]|
    Stage [IntT,IntF,OutT,OutF]                       (* second stage *)
endproc
```

A possible implementation for a FIFO stage is given in figure 2 (a). Apart from the data path, there are two lines that control data transmission. *Req* comes from the environment of a stage, indicating that environment has valid data to transfer. The *Ack* line goes to the environment, indicating that the stage is empty and is thus ready to receive new data. Both of these control signals are active when *1*. The implementation use two C-Elements (see section 2.3) and a *Nor2* gate (two-input 'not or'). Initially both *Req* and *Ack* are *1*. When there is valid data on *InT* or *InF*, it is passed to *OutT* or *OutF*. At the same time, *Req* should be reset to *0* until *InT* or *InF* returns to the idle state. *Ack* is reset to *0* after receiving data on *OutT* or *OutF*, indicating that the stage is full. When the data on output lines is fetched, the stage returns to the idle state and is ready for the next transmission. The corresponding DILL specification of this FIFO cell is as follows:

```
process Cell [InT,InF,OutT,OutF,Req,Ack] : noexit :=
  (CElement [InT,Req,OutT] (0,1,0) |[Req]| CElement [InF,Req,OuTF] (0,1,0))
|[OutT,OutF]|
  Nor2 [OutT,OutF,Ack] (0,0,1)
endproc
```

The inputs/output of the C-Elements are initialised to *0,1/0*, while those for the *Nor2* gate start as *0,0/1*.

To ensure a FIFO works correctly, the environment has to be coordinated. For example, it should provide correct input data according to the dual rail encoding. To make things easier, it is convenient to think about the environment in two parts: *EnvF* (environment front-end) is a provider that is always ready to produce data, while *EnvB* (environment back-end) is a consumer that can always accept data.

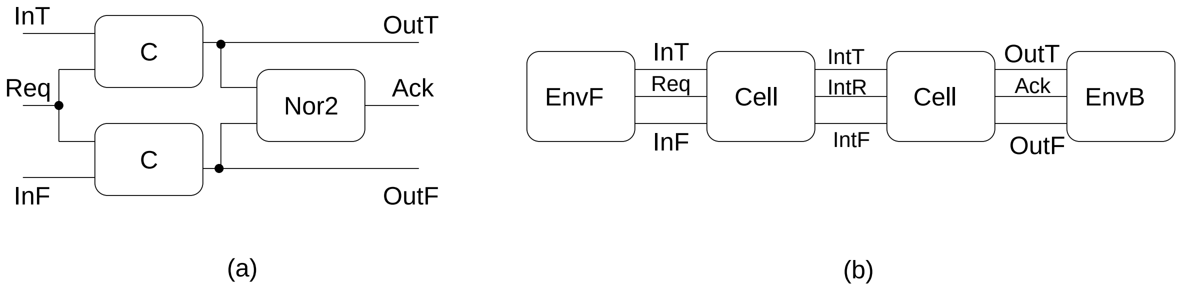


Figure 2: Implementation of A Two-Stage FIFO from Individual Cells

```
process EnvF [Req,InT,InF] : noexit :=            (* data provider *)
  InT !1; Req !0; InT !0; Req !1;                 (* provide 1 *)
  EnvF [Req,InT,InF]                              (* continue behaviour *)
[]
  InF !1; Req !0; InF !0; Req !1;                 (* provide 0 *)
  EnvF [Req,InT,InF]                              (* continue behaviour *)
endproc

process EnvB [Ack,OutT,OutF] : noexit :=          (* data consumer *)
  OutT !1; Ack !0; OutT !0; Ack !1;               (* accept 1 *)
  EnvB [Ack,OutT,OutF]                            (* continue behaviour *)
[]
  OutF !1; Ack !0; OutF !0; Ack !1;               (* accept 0 *)
  EnvB [Ack,OutT,OutF]                            (* continue behaviour *)
endproc
```

A two-stage FIFO can then be implemented as in figure 2 (b):

```
process Impl [InT,InF,OutT,OutF] : noexit :=      (* FIFO implementation *)
  hide Req,IntT,IntF,IntR,Ack in                  (* internal signals *)
    EnvF [Req,InT,InF]                            (* data provider *)
  |[Req,InT,InF]|
    Cell [InT,InF,IntT,IntF,IntR,Req]             (* first cell *)
  |[IntT,IntF,IntR]|
    Cell [IntT,IntF,OutT,OutF,Ack,IntR]           (* second cell *)
  |[Ack,OutT,OutF]|
    EnvB [Ack,OutT,OutF]                          (* data consumer *)
endproc
```

When speed independence needs to be verified, each building block (including the environment) should be specified in the input quasi-receptive style. *Impl_QR* is the corresponding implementation specification. It can use the DILL library for quasi-receptive specifications of the basic building blocks. It also needs the corresponding quasi-receptive specifications *EnvF_QR* and *EnvB_QR*. *EnvF* has no inputs and so is identical to *EnvF_QR*. *EnvB_QR* has the form:

```
process EnvB_QR [Ack,OutT,OutF] : noexit :=       (* receptive data consumer *)
  OutT !1;                                        (* value 1 output by FIFO *)
  (Ack !0;       (OutT !0;      (Ack !1; EnvB_QR [Ack,OutT,OutF]
                                [] OutT !1; stop [] OutF !1; stop)
                 [] Ack !1; stop [] OutF !0; stop)   (* incorrect ack or FIFO output *)
  [] OutT !0; stop [] OutF !0; stop)              (* incorrect FIFO output *)
[]
  OutF !1;                                        (* value 0 output by FIFO *)
  (Ack !0;       (OutF !0;      (Ack !1; EnvB_QR [Ack,OutT,OutF]
                                [] OutT !1; stop [] OutF !1; stop)
```

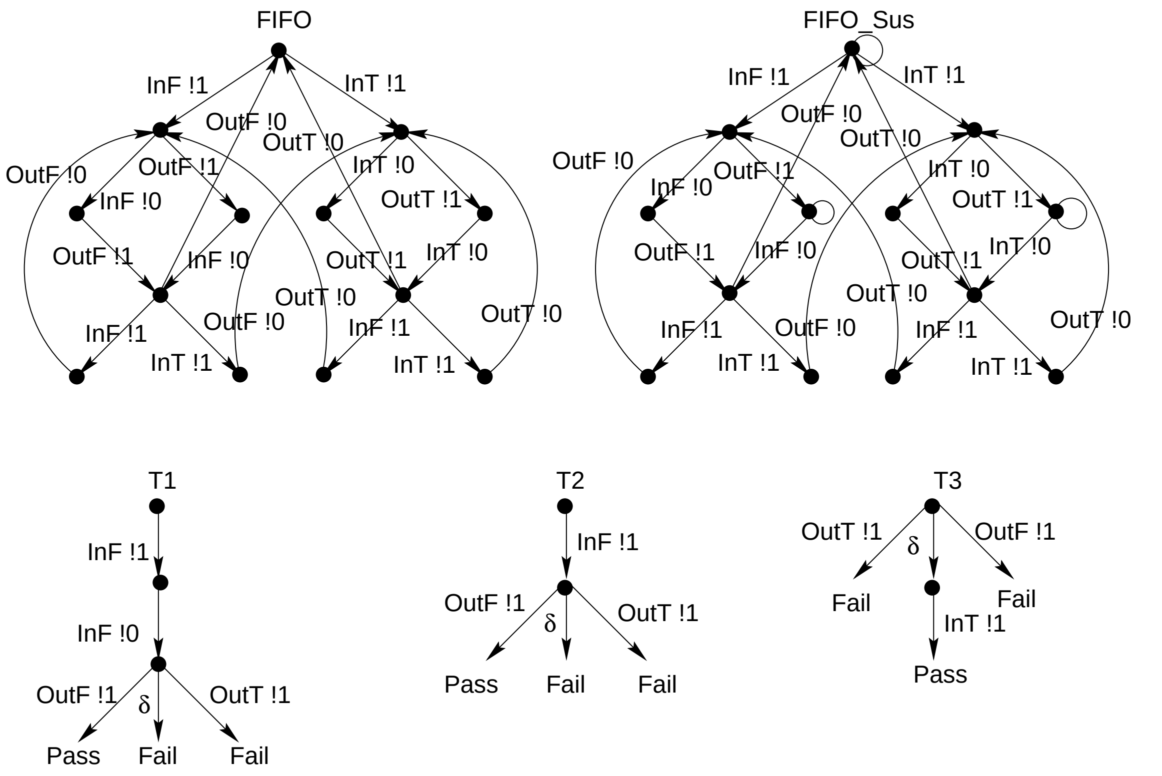


Figure 3: LTS, Suspension Automaton and Several Tests of FIFO

```
                           [] Ack !1; stop [] OutT !0; stop)        (* incorrect ack or FIFO output *)
    [] OutT !0; stop [] OutF !0; stop)                              (* incorrect FIFO output *)
  []
    Ack !0; stop
  endproc
```

The specification should exhibit liveness. Using CADP, it was verified that the specification satisfies the following property expressed in ACTL (Action-based Computational Tree Logic [12]). If there is an input of *1*, then output will become *1* eventually: $AG([\text{InT !1}]A[\text{true}_{true}U_{Out!1}\text{true}])$. The formula for data *0* is similar and was also verified to be true. It was verified that *Spec* $\approx$ *Impl* $||$ *(EnvB* $|[\cdots]|$ *EnvF)* using CADP, where $\approx$ denotes observational equivalence.

To check speed independence, the input quasi-receptive specifications were used. It was also verified that *Spec* $\approx$ *Impl_QR* $||$ *(EnvB_QR* $|[\cdots]|$ *EnvF_QR)*, which gives more confidence in the design of the FIFO. The implementation *Impl_QR* $||$ *(EnvB_QR* $|[\cdots]|$ *EnvF_QR)* also satisfies the liveness property.

Figure 3 gives the LTS for the FIFO (minimised with respect to observational equivalence), the suspension automaton for the LTS, and several tests. Because the LTS is deterministic, the suspension automaton has almost the same structure except for the $\delta$ transitions, which appear as circles in the figure. Test *T1* provides two inputs and then checks the output of an implementation. If output *OutF* changes, the implementation passes the test. However if *OutT* changes or if there is no output, the implementation fails the test. Similarly, test *T2* checks output after one input is provided. Test *T3* checks output right away. For this test, an output from the initial state is incorrect and results in a fail verdict. Only after a $\delta$ transition, meaning that no output is produced, can testing continue.

It was established that *Impl_QR* $||$ *(EnvB_QR* $|[\cdots]|$ *EnvF_QR) strongconf Spec* by using the authors' *VeriConf* tool. The authors' *TestGen* tool produces a single test case of length 28 for the FIFO:

| | | | | | | |
|---|---|---|---|---|---|---|
| InF !1 | InF !0 | OutF !1 | InF !1 | OutF !0 | OutF !1 | InF !0 |
| InT !1 | OutF !0 | InT !0 | OutT !1 | InT !1 | OutT !0 | OutF !1 |
| $\delta$ | InF !0 | OutF !0 | InT !1 | OutT !1 | InT !0 | InT !1 |
| OutT !0 | OutT !1 | $\delta$ | InT !0 | OutT !0 | $\delta$ | Pass |

## 5 Case Study: An *Or-And* Circuit

This example was introduced in [3] to show the difference between speed independence and delay insensitivity. Although small, it reveals the necessity of using input quasi-receptive specifications. As in section 2.3, events are abbreviated by omitting signal values *!1* and *!0*. The circuit has inputs *Ip1, Ip2, Ip3* and outputs *Op1, Op2*. Output *Op1* is the logical *or* of *Ip1, Ip2*, while output *Op2* is the logical *and* of *Ip2, Ip3*.

The abstract circuit specification is shown in figure 4 (a). The component *Or* and *And* gates are specified in figures 4 (b) and 4 (c) respectively. The proposed implementation is in figure 4 (d). The verification task is to check if this implementation is delay-insensitive and speed-independent. For analysing delay insensitivity, the circuit is transformed to figure 4 (e), where the isochronic fork in figure 4 (d) is replaced by an explicit Fork element. (Refer back to section 2.3 for an explanation of these.) The implementations in figures 4 (d) and figure 4 (e) are specified as *Impl1* and *Impl2*:

```
process Impl1 [Ip1,Ip2,Ip3,Op1,Op2] : noexit :=       (* isochronic fork implementation *)
  Or2 [Ip1,Ip2,Op1] |[Ip2]| And2 [Ip2,Ip3,Op2]         (* or plus and gates *)
endproc

process Impl2 [Ip1,Ip2,Ip3,Op1,Op2] : noexit :=       (* ordinary fork implementation *)
  hide Int1,Int2 in
    (Or2 [Ip1,Int1,Op1] ||| And2 [Int2,Ip3,Op2])       (* or plus and gates ... *)
  |[Int1,Int2]|
    Fork [Ip2,Int1,Int2]                               (* with fork input *)
endproc
```

The state spaces of *Impl1* and *Impl2* are much larger than that of *Spec*. For example, both can accept *Ip2* and *Ip3* from their initial states but *Spec* cannot. Since no explicit environment is given, a direct verification approach is to compare *Impl1* || *Spec* with *Spec*, i.e. assuming that *Spec* is also the environment of its implementations. It was found that *Impl1* || *Spec* $\approx$ *Spec* and *Impl2* || *Spec* $\approx$ *Spec* by using CADP. This suggests that figures 4 (d) and (e) are both correct implementations of *Spec*.

However, to ensure that both implementations are really speed-independent, the more accurate input quasi-receptive model is needed. Figure 5 shows the revised LTSs. The corresponding implementations are *Impl1_QR* and *Impl2_QR*. It was discovered that *Impl1_QR strongconfor Spec* by using the authors' *VeriConf* tool. Unfortunately *Impl2_QR* does not have the *confor* or *strongconfor* relationship to *Spec*. The tool gives a diagnostic trace: *Ip1, Op1, Ip3, Ip2, Op2, Ip2, Ip1*. By analysing this trace it can be found that after *Op2* is produced, the specification is able to receive *Ip1* and *Ip2*. But for the implementation in figure 4 (e), after *Op2* is produced the fork component may still be in the unstable state since *Int1* has not been produced yet. In this unstable state, an *Ip2* input makes the behaviour of the fork component undefined. This means that figure 4 (e) is not speed-independent, i.e. the correctness of the circuit depends on the speed of *Fork*. Figure 4 (d) is therefore not a delay-insensitive implementation of the specification.

## 6 Case Study: A Selector

As a final example, a selector (see section 2.3) allows non-deterministic behaviour in implementations. After a change on input *Ip*, either *Op1* or *Op2* may change depending on the implementation. Figure 6

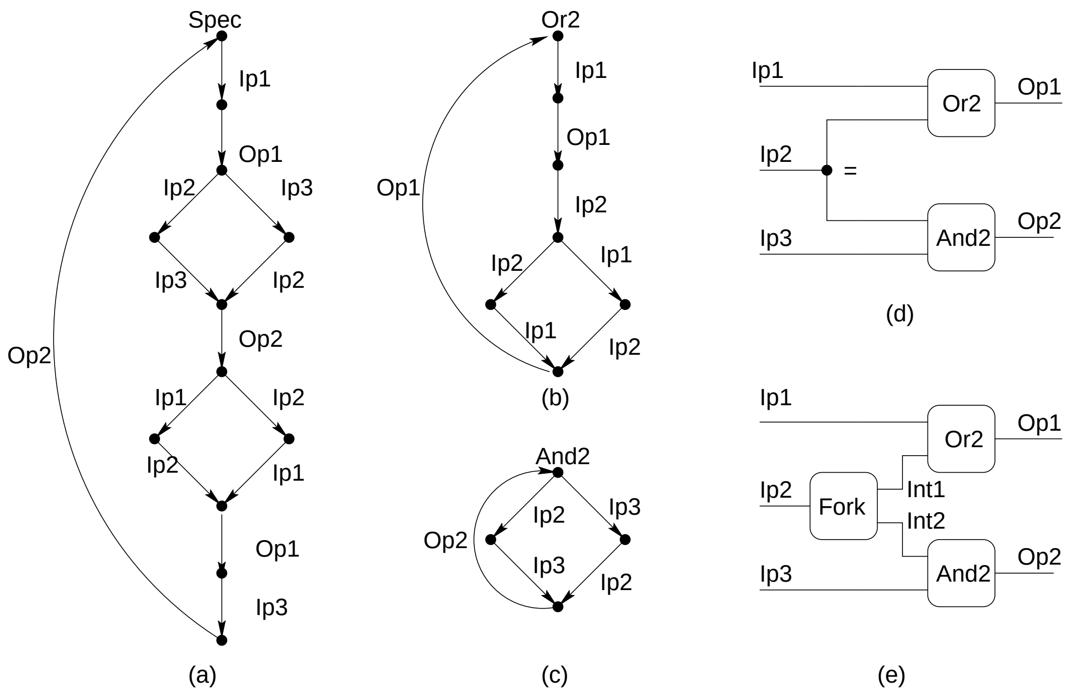


Figure 4: An *Or-And* Circuit

gives its LTS (minimised with respect to observational equivalence), the suspension automaton of the LTS, and one of the test cases. Test *T4* shows that after input *Ip !1*, an implementation producing either *Op1 !1* or *Op2 !1* will pass the test.

The authors' *TestGen* tool produces a single test case of length 11 for the selector. This example shows how contradictory branches are marked. A selector that insists on sending its input to *Op1* can follow test steps *1, 2, 3, 4, 5, 6, 2, 3, ...* This is a loop that the testbench must break.

| | | | | | |
|---|---|---|---|---|---|
| IP !1 | Op1 !1 ($\star$S1) | Ip !0 | Op1 !0 ($\star$S2) | $\delta$ | Ip ! 1 |
| Op2 !1 ($\star$S1) | $\delta$ | Ip !0 | Op2 !0 ($\star$S2) | Pass | |

# 7 Conclusion

An approach to verifying asynchronous circuits has been presented. A key aim has been to model real hardware effectively using LOTOS. A range of typical asynchronous components has been specified, demonstrating the applicability of the approach. (Quasi) delay-insensitive circuits are transformed into speed-independent designs. Violations of speed-independence (or really semi-modularity) are checked using input (quasi-)receptive specifications.

New relations *confor* and *strongconfor* have been defined to assess the implementation of an asynchronous circuit against its specification. The relations provide an intuitive interpretation of correctness and offer clear advantages compared to other approaches. The *VeriConf* tool has been developed to support them. The theory of Input-Output Labelled Transition Systems has also been adapted for generating tests of asynchronous circuits based on suspension automata. The *TestGen* tool generates test suites using transition tours of automata. This allows automatic generation of test suites with reasonable coverage, and facilitates automatic test execution. The test generation algorithm also allows non-deterministic implementations to be tested.

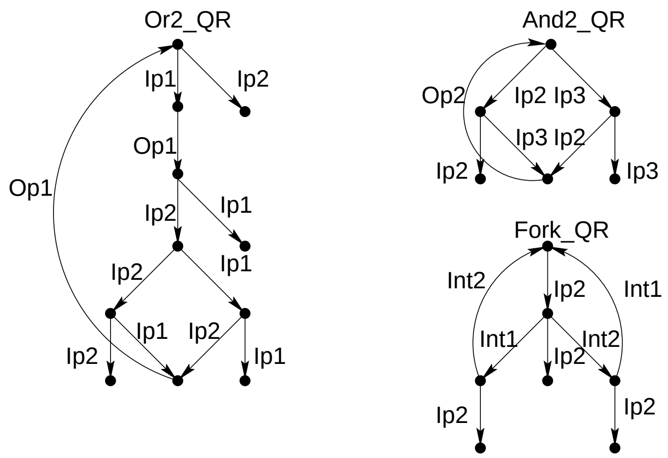


Figure 5: Input Quasi-Receptive Specifications of *Or-And* Components

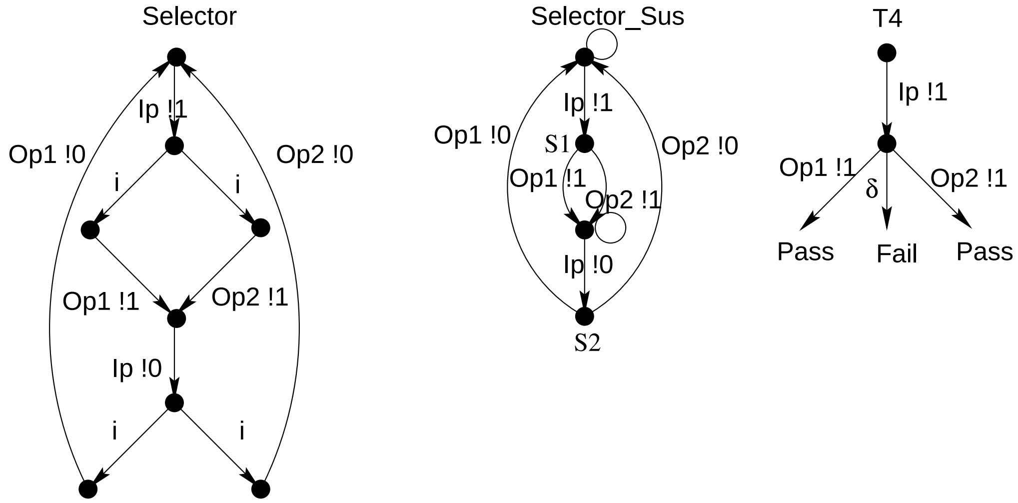


Figure 6: LTS, Suspension Automaton and One Test of Selector

# References


[1] A. Cerone, D. A. Kearney, and G. J. Milne. Integrating the verification of timing, performance and correctness properties of concurrent systems. In *Proc. Application of Concurrency to System Design*, pages 109–119. IEEE Computer Society Press, 1998.

[2] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.

[3] J. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, pages 107–119, 1991.

[4] J. C. Ebergen, J. Segers, and I. Benko. Parallel program and asynchronous circuit design. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.

[5] J.-C. Fernández, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDÉBARAN Development Package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proc. 8th. Conference on Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 437–440. Springer-Verlag, Berlin, Germany, Aug. 1996.

[6] G. Gopalakrishnan, E. Brunvand, N. Michell, and S. Nowick. A correctness criterion for asynchronous circuit validation and optimization. *IEEE Transactions on Computer-Aided Design*, 13(11):1309–1318, Nov. 1994.

[7] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proc. 22nd. Annual International Synposium on Computer Architecture*, 1995.

[8] IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.

[9] Ji He and K. J. Turner. Protocol-inspired hardware testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Proc. Testing Communicating Systems XII*, pages 131–147, London, UK, Sept. 1999. Kluwer Academic Publishers.

[10] Ji He and K. J. Turner. Specification and verification of synchronous hardware using LOTOS. In J. Wu, S. T. Chanson, and Q. Gao, editors, *Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)*, pages 295–312, London, UK, Oct. 1999. Kluwer Academic Publishers.

[11] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger. Synthesis of delay-insensitive modules. In H. Fuchs, editor, *Proc. Chapel Hill Conference on Very Large Scale Integration*, pages 67–86. Computer Science Press, 1985.

[12] R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. In *Proc. 5th. Annual Symposium on Logic in Computer Science (LICS 90)*, pages 118–129. IEEE Computer Society Press, 1990.

[13] J. M. T. Romijn, O. Sies, and J. R. Moonen. A two-level approach to automated conformance testing of VHDL designs. *Testing of Communicating Systems*, 10:432–447, 1997.

[14] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.