VERIFYING AND TESTING ASYNCHRONOUS CIRCUITS USING LOTOS

Ji He and Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

h.ji@reading.ac.uk, kjt@cs.stir.ac.uk

Abstract

It is shown how DILL (Digital Logic in LOTOS) can be used to specify, verify and test asynchronous (unclocked) hardware designs. 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 insensitivity and testing of sample asynchronous circuits.

Keywords:

Conformance Testing, Hardware Description, Hardware Verification, LOTOS

1. INTRODUCTION

DILL (Digital Logic in LOTOS, e.g. [8, 9]) is an approach, a language and a toolset for specifying, analysing and testing digital hardware designs. DILL builds on the theories and tools for LOTOS. The authors have previously developed methods for dealing with synchronous (clocked) circuits. The recent work reported in this paper extends DILL to asynchronous (unclocked) circuits.

Conventional digital circuits operate under control of a master clock for synchronisation, avoiding problems due to component speed variations. However, asynchronous circuits are interesting as they operate at the speed of individual components. Unfortunately asynchronous circuits are much harder to design and analyse. Verification of asynchronous circuits, especially speed-independent and delay-insensitive ones, has been a research topic for some years [1, 2]. Rigorous testing of asynchronous circuits is still in its infancy.

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*

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35533-7_26](https://doi.org/10.1007/978-0-387-35533-7_26)

[1], *decomposition* [2] and *strong conformance* [5]. It is not possible to detect deadlocks and livelocks with *conformance* and *decomposition*. Although *strong conformance* can do this, it does not work for non-deterministic specifications.

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. In the DILL approach, tests are derived from higher-level specifications in a novel adaptation of the conformance testing theory developed for protocols.

2. CIRCUIT SPECIFICATION

2.1. ASYNCHRONOUS CIRCUIT CLASSES

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

DI (Delay-Insensitive) circuits assume unbounded (but finite) delays for components and wires. Correct operation does not depend on actual delays.

QDI (Quasi Delay-Insensitive) circuits augment the delay model of DI 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 with zero delay. If all gates have just one output, SI and QDI are 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 wire delays can be absorbed into the preceding components. Only 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 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, a basic logic gate is used to illustrate how to model speed independence (really semi-modularity) of asynchronous circuits. In [8, 9], 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, the 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 at 0. This depends on the speed of the gate. To respect semi-modularity, such inputs have to be forbidden. Note that the specification is partial in that inputs are forbidden at certain points. An input offer happens only when there is no potential output, or when input cannot alter potential output.

```

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 *)
  ]                                                                    (* or *)
  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 *)
        ((dtIp1 nand newIp2) eq newOp))]);                        (* but no change is needed *)
    Nand2 [Ip1,Ip2,Op] (dtIp1,newIp2,dtOp)                          (* continue behaviour *)
  ]                                                                    (* or *)
  Op !newOp [dtOp ne newOp];                                         (* new output produced *)
  Nand2 [Ip1,Ip2,Op] (dtIp1,dtIp2,newOp)                          (* continue behaviour *)
  )
endproc

```

2.3. SPECIFYING CIRCUIT COMPONENTS

Asynchronous circuit design uses special-purpose components in addition to basic logic gates. 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 to omit signal levels when specifying asynchronous circuits, since the levels strictly alternate. For example the input

sequence $lp !0$, $lp !1$, $lp !0$ can be represented as lp , lp , lp . The following outline specifications respect semi-modularity.

Wires are the simplest components. Delays on wires are assumed zero, so they are not needed for speed-independent circuits. But when a (quasi) delay-insensitive circuit is made speed-independent, wire delays may have to be explicitly specified.

Wire $[Ip, Op] := Ip; Op; \text{Wire } [Ip, Op]$

Fork components are also necessary when a (quasi) delay-insensitive design is transformed to speed-independent form. A fork has one input lp and two outputs $Op1$, $Op2$. The value on input lp 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 $[Ip, Op1, Op2] := Ip; (Op1; \text{exit} ||| Op2; \text{exit}) \gg \text{Fork } [Ip, Op1, Op2]$

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 $lp1$, $lp2$ and an output Op . The output changes to 1 when both inputs change to 1, and changes to 0 when both change to 0.

C-Element $[lp1, lp2, Op] := (lp1; \text{exit} ||| lp2; \text{exit}) \gg (Op; \text{C-Element } [lp1, lp2, Op])$

Merge components copy input signals $lp1$, $lp2$ to a single output Op .

Merge $[lp1, lp2, Op] := lp1; Op; \text{Merge } [lp1, lp2, Op] \sqcap lp2; Op; \text{Merge } [lp1, lp2, Op]$

Selectors take an input lp , and non-deterministically output on $Op1$ or $Op2$.

Selector $[lp, Op1, Op2] := lp; (i; Op1; \text{exit} \sqcap i; Op2; \text{exit}) \gg \text{Selector } [lp, Op1, Op2]$

2.4. INPUT (QUASI-)RECEPTIVENESS

In LOTOS, communication between processes is based on symmetric synchronisation at a gate. However, digital hardware has a clear distinction between inputs and outputs. A hardware component can never refuse inputs, while its outputs can never be blocked by other components.

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. 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 specification is input quasi-receptive if it can always participate in input events except when deadlocked. An input quasi-receptive specification can be obtained by adding a choice when there is a potential output.

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 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 $lp1$, performs an internal action, and ends up in state S' where $lp2$ can be accepted. It would be incorrect to add a transition to state S that accepts $lp2$ and leads to deadlock. That is, S accepts $lp2$ only through an internal action.

Internal events are less meaningful when considering input receptiveness as they mean the environment has no effect on choices. For this reason, LTSs with internal events are determined 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. CIRCUIT VERIFICATION AND TESTING

3.1. VERIFYING CIRCUIT DESIGNS

As it is more difficult to specify components in an input (quasi-)receptive manner, verification may be based on 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.

Assumptions about the environment often have to be made. When an environment is not explicit, many methods simply assume the mirror of a specification as the environment of its implementations [1]. If S is the abstract specification and I is the implementation specification, verification means comparing $S \parallel I$ with S or checking if a logic formula holds for $S \parallel I$. But verifying $S \parallel I$ is not always satisfactory. When an implementation can accept more inputs than its specification does, $S \parallel I$ 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.

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 a (quasi-)receptive environment or implementation, and are natural criteria for asynchronous circuit correctness.

3.2. INPUT-OUTPUT CONFORMANCE

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. The inspiration was *ioconf* and *ioco* [11] for protocol conformance testing.

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 a circuit may accept all inputs at any time, most specifications are partial to avoid 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, consider the input transitions of *sp* and *im*. If input *ip* is accepted by *sp*, it is reasonable that *ip* also be accepted by *im*. But if *im* accepts an input that is not accepted by *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*.

Concerning outputs, 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 δ ‘action’ overcomes this weakness by indicating the absence of output. Like any other output, if δ is in the output set of *im* it must be in the output set of *sp* for conformance to hold. That is, *im* can produce nothing only if *sp* can do nothing.

In the above, *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

δ is also involved in the sequence, the state spaces of both specification and implementation are transformed into automata that are explicitly labelled with δ . The input-output sequences are actually traces of the specification automaton. For implementation $i \in \mathcal{LTS}(L_I \cup L_U)$ and specification $s \in \mathcal{LTS}(L_I \cup L_U)$:

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

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 output refusals: self-loop transitions labelled with L_U indicate that no output action can occur. Refusal to output can also be expressed using δ , which is regarded as an output action distinct from L_I and L_U . A suspension trace thus contains ordinary actions and δ actions. If L_δ denotes $L \cup \delta$, a suspension trace $\sigma \in L_\delta^*$.

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 for a deterministic specification and implementation, while *strongconfor* is used when an implementation is more deterministic than a specification. (*strongconfor* is more easily observed if a specification LTS is transformed to a suspension automaton.

A suspension automaton Γ_p of an LTS p is obtained by determinising p and adding the necessary δ transitions. The suspension traces of p coincide with the traces of its suspension automaton Γ_p . In addition, for all $\sigma \in L^*$, $\text{out}(\Gamma_p \text{ after } \sigma) = \text{out}(p \text{ after } \sigma)$; see [11] for the proof. Therefore checking (*strongconfor*) can be easily reduced to checking trace inclusion on suspension automata. Only traces without δ transitions are checked for *confor*, while all the traces of a suspension automaton are checked for *strongconfor*.

A verification tool *VeriConf* was developed by the authors to check the (*strongconfor*) relations using the programming interface of CADP (Csar Aldbaran Development Package [4]). 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

By ‘test hypothesis’, an implementation is assumed to be modelled by an IOLTS (Input Output Labelled Transition System) with actions L partitioned into inputs L_I and outputs L_U . Input actions are always enabled in any state. This class of system is denoted by $\text{IOLTS}(L_I, L_U) \subseteq \text{LTS}(L_I \cup L_U)$. Specifications, however, are still modelled as LTSs to permit an abstract view. They are interpreted as incompletely specified IOLTSs where some inputs are not specified in some states. Incompleteness might be for 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 [11] has been defined to support conformance testing; it 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 with finite behaviour. Test cases must also be deterministic to allow a tester to have control over test execution. 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 δ action). The terminal states of a test are labelled with *Pass* or *Fail* to yield a verdict. 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 $\text{LTS}(L_I \cup L_U \cup \{\delta\})$. As before, δ 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 $\text{out}(T)$. This test generation algorithm is sound and complete with respect to (*strong*)*confor*; see [11] 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 [7]). The behaviour of a VHDL program is presumed to be modelled by an IOLTS that need not be known explicitly. The test suite for a circuit is generated

by an algorithm based on that of [11]. 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.

The test cases generated in the 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 a suspension automaton may not be strongly connected, it is not possible to make direct use of conventional transition tour algorithms. Instead the approach of [6] 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. 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.

Figure 1 is an example of this technique. If an implementation has the behaviour $Ip, Op1, Op2, Op3, \dots$ 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 $Ip, Op3, \dots$ 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. If there is no output within a certain period, the δ transition can be regarded as having occurred; otherwise a failure verdict must be given. A testbench

A possible test trace:

$Ip, Op1(*S1), Op2, Op1(*S4), \dots$

$Op3(*S4), Ip, Op2(*S1), Op1\dots$

Figure 1 Test Trace for Nodes with Several Outputs

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*.

4. CASE STUDY: 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 raised input and output lines return to *0* to wait for other data. One FIFO stage is specified as:

```

process Stage [InT,InF,OutT,OutF] :noexit :=
    InT !1; OutT !1; InT !0; OutT !0;          (* one FIFO stage *)
    Stage [InT,InF,OutT,OutF]                  (* input/output 1 then idle *)
    Stage [InT,InF,OutT,OutF]                  (* continue behaviour *)
    InF !1; OutF !1; InF !0; OutF !0;          (* or *)
    Stage [InT,InF,OutT,OutF]                  (* input/output 0 then idle *)
    Stage [InT,InF,OutT,OutF]                  (* continue behaviour *)
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 uses 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* when ready to send data on *OutT* or *OutF*, indicating the stage is full. When the data on output lines is fetched, the stage returns to idle and is ready for the next data. The inputs/output of the C-Elements are start as *0,1/0* while those for the *Nor2* gate start as *0,0/1*. The specification of this FIFO cell is:

```

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

```

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* (front-end) is a provider that is always ready to produce data, while *EnvB* (back-end) is a consumer that can always accept data.

[illegible][illegible]

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 [InT, InF, Req] (* data provider *)  
  |[InT, InF, Req]| (* synchronised with *)  
    Cell [InT, InF, IntT, IntF, Req, IntR] (* first cell *)  
  |[IntT, IntF, IntR]| (* synchronised with *)  
    Cell [IntT, IntF, OutT, OutF, IntR, Ack] (* second cell *)  
  |[OutT, OutF, Ack]| (* synchronised with *)  
    EnvB [OutT, OutF, Ack] (* data consumer *)  
endproc
```

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

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 [OutT, OutF, Ack] : noexit :=      (* receptive data consumer *)
  OutT !1;                                         (* value 1 output by FIFO *)
  (Ack !0;      (OutT !0;      (Ack !1; EnvB_QR [OutT, OutF, Ack]
    [] 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 *)
  []                                              (* or *)
  OutF !1;                                         (* value 0 output by FIFO *)
  (Ack !0;      (OutF !0;      (Ack !1; EnvB_QR [OutT, OutF, Ack]
    [] OutT !1; stop [] OutF !1; stop)
    [] 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 [10]). If there is an input of 1, then output will become 1 eventually: $AG([InT !1]A[true_{true}U_{Out!1}true])$. The formula for data 0 is similar and was also shown to be true. It was verified that $Spec \approx Impl \parallel (EnvB \parallel [\cdot \cdot] EnvF)$, where $\approx$ denotes observational equivalence.

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

Figure 3 LTS, Suspension Automaton and Several Tests of FIFO

same structure except for the δ 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. Only after δ , meaning that no output is produced, can testing continue.

It was shown that $Impl_QR \parallel (EnvB_QR \parallel [\cdot \cdot \cdot] EnvF_QR)$ *strongconfor* *Spec* using the *VeriConf* tool. The *TestGen* tool builds a single test case of length 28:

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
δ	InF !0	OutF !0	InT !1	OutT !1	InT !0	InT !1
OutT !0	OutT !1	δ	InT !0	OutT !0	δ	Pass

5. CASE STUDY: 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 *lp1*,

Figure 4 An Or-And Circuit

$Ip2$, $Ip3$ 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 LTS is shown in figure 4 (a). The component *Or* and *And* LTSs appear 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 (see sections 2.1 and 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 (* hide internal signals *)
    (Or2 [Ip1,Int1,Op1] ||| And2 [Int2,Ip3,Op2]) (* or plus and gates *)
    || [Int1,Int2] (* synchronised with *)
    Fork [Ip2,Int1,Int2] (* 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 $Impl1 \parallel Spec$ with *Spec*, i.e. assuming that *Spec* is also the environment of its implementations. It was found that $Impl1 \parallel Spec \approx Spec$

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

and $Impl2 \parallel Spec \approx Spec$ 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 *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.

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

Using the method in section 3.3, the *TestGen* tool produces a single test case of length 11 for the selector. This example shows how contradictory branches are marked: after $Ip !1$, $Op !1$ is marked with state $S1$ since an implementation

Figure 6 LTS, Suspension Automaton and One Test of Selector

may also do $Op\ !2$. A selector that insists on sending its input to $Op1$ can follow the first row of steps in the test case below. After the sixth step ($Ip\ !1$), it cycles back to the second step ($Op\ !1$) – a loop that the testbench must break.

$Ip\ !1$	$Op1\ !1\ (*S1)$	$Ip\ !0$	$Op1\ !0\ (*S2)$	δ	$Ip\ !1$
$Op2\ !1\ (*S1)$	δ	$Ip\ !0$	$Op2\ !0\ (*S2)$	Pass	

7. CONCLUSION

An approach to verifying asynchronous circuits has been presented for a range of typical asynchronous components. (Quasi) delay-insensitive circuits are transformed into speed-independent designs. Violations of speed-independence (or rather of semi-modularity) are checked using specifications that are input (quasi-)receptive. New relations *confor* and *strongconfor* have been defined to assess the implementation of an asynchronous circuit against its specification. The *VeriConfor* tool has been developed to support the (*strong*)*confor* relations. The *TestGen* tool generates test suites using transition tours of automata. This allows automatic generation of test suites for reasonable coverage, and also allows testing of non-deterministic implementations.

References

- [1] D. L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*. ACM Distinguished Dissertations. MIT Press, 1989.
- [2] J. C. Ebergen, J. Segers, and I. Benko. Parallel program and asynchronous circuit design. In G. Birtwistle and A. Davis, editors, *Asynchronous Dig-*

- ital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.
- [3] J. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, pages 107–119, 1991.
- [4] 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.
- [5] 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.
- [6] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proc. 22nd. Annual International Symposium on Computer Architecture*, 1995.
- [7] IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.