Communicating Systems XII, pages 131-147, Kluwer Academic Publishers, London, UK, September 1999.

1

PROTOCOL-INSPIRED HARDWARE TESTING

Ji He and Kenneth J. Turner
Computing Science and Mathematics, University of Stirling, Scotland FK9 4LA
jih@cs.stir.ac.uk, kjt@cs.stir.ac.uk

Abstract

The relevance of protocol conformance testing techniques to hardware testing is discussed. It is shown that the *ioconf* (input-output conformance) approach used in protocol testing can be applied to generate tests for a synchronous hardware design using its formal specification. The generated tests are automatically applied to a circuit by a VHDL testbench, thus giving confidence that the hardware design meets its high-level formal specification. Case studies illustrate how the ideas can be applied to standard hardware verification benchmarks such as the Single Pulser and Black-Jack Dealer.

Keywords:

Conformance Testing, Design Validation, Hardware Description

1. INTRODUCTION

1.1 BACKGROUND

Modern digital circuits are becoming extremely complex, requiring substantial effort to ensure design correctness prior to manufacture. DILL (Digital Logic in LOTOS [14, 15, 16, 25]) is an approach and a language for specifying digital circuits using LOTOS (Language of Temporal Ordering Specification [12]). The formal basis of LOTOS supports rigorous specification and analysis, both crucial for correct circuit design.

The inspiration for the work presented in this paper comes from conformance testing of communications protocols. The paper thus concerns testing of communicating systems, but in this case hardware devices rather than protocol entities. In formally based protocol testing, tests are derived from a formal specification and then used to check a concrete implementation. This is regarded as a black box whose operation has to be checked against its specification. Essentially this is the way that the functionality of a digital circuit is tested. Moreover, as in a communications system each part of a circuit has to communicate with the other parts. As a novel application of conformance testing, it is therefore worthwhile investigating how protocol testing ideas can be applied to validation of digital hardware.

In electronics, the equivalent term is **design verification**, **design validation** or **functional testing**. The term testing in this area applies to checking *manufacturing* defects in products rather than *design* problems. In the field of formal methods, the term testing has a more general meaning. The system under test might be a physical product, a formal specification or an informal specification. This paper interprets testing to mean evaluation of a requirements specification against a purported implementation specification.

1.2 APPROACH

Conformance testing uses experimentation to check an implementation against its formal specification. Tests are derived from the specification, then applied to the **IUT** (Implementation Under Test). Based on observations made during the execution of the tests, a verdict is given about the correct functioning of the implementation.

Hardware circuits are specified in this paper using LOTOS, whose semantics is given by an **LTS** (Labelled Transition System). The implementation of the same circuit is described by VHDL (VHSIC Hardware Description Language [10]). The behaviour of a VHDL program is presumed to be modelled by an **IOLTS** (Input-Output Labelled Transition System). This model is merely assumed to exist – it need not be known explicitly. Making the assumption that an implementation has a formal model is referred to as the **test hypothesis**. This makes it possible to express conformance of an implementation with respect its specification using a formal relation. One such relation, *ioconf* (input-output conformance [24]), is used as the criterion for correct hardware design.

The test suite for a circuit is generated from a LOTOS specification following an algorithm based on that proposed by Tretmans [24]. The authors have extended CADP (Cæsar Aldébaran Development Package [6]) to generate hardware test suites automatically. Each test case in the generated test suite is a sequence of input and output signals. Designing test cases as input-output sequences is close to engineering practice in hardware testing. Moreover, it allows test execution and obtaining test verdicts to be completely automated. This is achieved by a VHDL testbench that executes and evaluates the test cases. If there is an inconsistency between the formal specification and its VHDL implementation, the implementation is regarded as incorrect.

Section 2. introduces the theory for testing an IOLTS. This is followed by its application to testing digital circuits in Section 3. Section 4. examines the techniques in two case studies.

1.3 RELATED WORK

Test theories for LTSs were first studied more than a decade ago. These theories aim to define implementation relations by explicitly using external tests and observations (e.g. [4, 19]). Apart from defining an implementation relation, conformance testing involves finding a set of tests for a specification to distinguish between correct and incorrect implementations. [2] elaborates a theory for testing systems specified in LOTOS. Several test generation algorithms for an LTS and for Basic LOTOS have been proposed, e.g. [17, 20]. In [23, 24] the testing theory for an LTS is refined for communicating systems that distinguish inputs and outputs. This is a more realistic view of systems.

For validating hardware designs, simulation has been and is still the predominant method in industry. In the main, test cases for simulation are manually defined or are randomly generated. Recent developments for improving this *ad hoc* approach lie in combining formal methods with traditional simulation techniques. In [26] design verification tests are generated from behavioural VHDL programs using traditional software testing methods. In [8, 18] test generation is based on an FSM (Finite State Machine) or ECFM (Extracted Control Flow Machine) that represents the control logic of a circuit. The generated test cases are then applied to both higher level and lower level specifications in Verilog [11] or VHDL. Verdicts are obtained by comparing outputs from the two levels. These approaches *extract* a formal model from a circuit design and use techniques based on FSM testing theory. But in this paper, tests are *derived* from higher level specifications using conformance testing theory for LTSs.

In [21] test generation is based on a higher level FSM specification using a commercial tool. Tests are then applied using a VHDL simulator. Unfortunately this method cannot handle non-determinism in specifications. Its aim is to fill the gap between abstract tests and concrete test signals. Of direct relevance to the current work, the CADP toolset has a test generation tool TGV [7] under development. The implementation relation exploited by TGV is very similar to *ioconf* used in this paper. TGV was still to be released at the time of writing, and so was not available to the authors for evaluation.

2. TESTING IO TRANSITION SYSTEMS

2.1 IOLTS AND IOCONF

Conformance testing involves checking the correctness of an implementation against its specification by external tests and observations. To formally define an implementation relation, a test hypothesis is needed that implementations can be expressed by a formal model. In traditional conformance testing theories for LTSs, both the specification and the IUT are modelled as LTSs. An IUT communicates with its environment through symmetric interactions, hence its environment as expressed by tests is also modelled as an LTS.

An **LTS** is a quadruple $\langle S, L, T, s0\rangle$ where *S* is a set of states, *L* is a set of observable actions, $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, and $s0 \in S$ is the initial state. The class of transition systems with actions in *L* is denoted by $\mathcal{LTS}(L)$. A transition in *T* is also denoted as $s \xrightarrow{\mu} s'$ if $(s, \mu, s') \in T$. The special action $\tau \notin L$ represents an unobservable (or internal) action. The following notations are commonly used for LTSs.

Let $p = \langle S, L, T, s0\rangle$ be an LTS with $s, s' \in S$, and let $\mu_i \in L \cup \{\tau\}$, $a_i \in L$, $L^\star$ denotes the set of all finite action sequences of *L* and $\sigma \in L^\star$. The following definitions then apply:

$$\begin{array}{rcl}
s \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} s' \\
s \overset{\mu_1\cdot\ldots\cdot\mu_n}{\not\longrightarrow} & =_{def} & \nexists s' : s \xrightarrow{\mu_1\cdot\ldots\cdot\mu_n} s' \\
s \overset{\epsilon}{\Rightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau\cdot\ldots\cdot\tau} s' \\
s \overset{a}{\Rightarrow} s' & =_{def} & \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \overset{a}{\rightarrow} s_2 \overset{\epsilon}{\Rightarrow} s' \\
s \xRightarrow{a_1\cdot\ldots\cdot a_n} s' & =_{def} & \exists s_0 \ldots s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s' \\
s \overset{\sigma}{\Rightarrow} & =_{def} & \exists s' : s \overset{\sigma}{\Rightarrow} s' \\
s \overset{\sigma}{\not\Rightarrow} & =_{def} & \nexists s' : s \overset{\sigma}{\Rightarrow} \\
\mathit{init}(p) & =_{def} & \{\mu \in L \cup \{\tau\} \mid p \overset{\mu}{\rightarrow}\} \\
\mathit{traces}(p) & =_{def} & \{\sigma \in L^\star \mid p \overset{\sigma}{\Rightarrow}\} \\
\text{p } \mathit{after}\ \sigma & =_{def} & \{p' \mid p \overset{\sigma}{\Rightarrow} p'\}
\end{array}$$

Many real world systems communicate with their environment in a different way from an LTS. There is a clear distinction between the inputs and outputs of a system. The inputs of a system are always enabled and cannot refuse the actions offered by the environment. After the system consumes an input and produces its outputs, the environment has to accept the outputs. In other words, such a system will never reject inputs and its environment will never block outputs. Communication is thus no longer symmetric. In [24] this kind of behaviour is modelled as an IOLTS, which is a special kind of LTS.

An **IOLTS** (Input-Output Labelled Transition System) *p* is an LTS in which the set of actions *L* is partitioned into input actions L_I and output actions L_U such that $L_I \cup L_U = L$ and $L_I \cap L_U = \emptyset$. (The suffix $_U$ derives from the Dutch/German word for out.)

Whenever $p \xRightarrow{\sigma} p'$ then $\forall a \in L_I : p' \xRightarrow{a}$

Intuitively this means that input actions are always enabled in any state. The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOLTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$.

Several implementation relations have been defined to express conformance of an implementation to its specification. In these relations, specifications are modelled as LTSs and implementations are modelled as IOLTSs. This is because an LTS can give a more abstract view of a system, while an IOLTS is closer to reality. The specification LTS can be regarded as a partially specified IOLTS in the sense that there are some states in the specification that can refuse input actions. There are two reasons to write such kinds of specifications. One is that it does not matter how implementations respond to unspecified inputs. The other is that the environment is assumed not to offer such inputs, so there is no need to specify them.

To define the implementation relation *ioconf*, several other definitions have to be introduced. Let $p \in \mathcal{LTS}(L_I \cup L_U)$, *s* be a state in the LTS and *S* be a state set. Then:

- A state *s* of *p* is **quiescent**, denoted by $\delta(s)$, if $\forall \mu \in L_U \cup \{\tau\} : \quad s \overset{\mu}{\not\longrightarrow}$
- A **quiescent trace** of *p* is a trace σ which may lead to a quiescent state: $\exists p' \in (p \ \mathit{after} \ \sigma) : \delta(p')$
- $\mathit{out}(s) \quad =_{def} \quad \{x \in L_U \mid s \xrightarrow{x}\} \cup \{\delta \mid \delta(s)\}$
- $\mathit{out}(S) \quad =_{def} \quad \bigcup\{out(s) \mid s \in S\}$

From the definition, a quiescent state is one that cannot perform any output transitions or an internal transition. *out(s)* defines all the output actions that a state can perform. This includes the quiescent 'action' δ that means the state cannot perform any output actions. Let $i \in \mathcal{IOLTS}(L_I, L_U), s \in \mathcal{LTS}(L_I \cup L_U)$. Then:

$$i \ \mathit{ioconf} \ s \quad =_{def} \quad \forall \sigma \in \mathit{traces}\ (s) : \ \mathit{out}(i \ \text{after} \ \sigma) \subseteq \mathit{out}(s \ \mathit{after} \ \sigma)$$

This means that an implementation is correct if, after all traces σ of the specification, the implementation outputs can also be produced by the specification. An implementation cannot produce outputs which are not expected by the specification. Since this also holds for δ, the implementation may not output if the specification cannot do so.

2.2 TEST GENERATION FOR IOCONF

To support the generation of test cases for *ioconf*, an intermediate LTS termed the **suspension automaton** is built from the specification LTS. The suspension automaton Γ_p of an LTS *p* is obtained by adding self-loops $s \xrightarrow{\delta} s$ for all quiescent states and then determinising the resulting automaton. The important properties of a suspension automaton are that it is deterministic and for $\sigma \in L^\star$, $out(\Gamma_p \text{ after } \sigma) = out(p \text{ after } \sigma)$. As will be seen later, checking *ioconf* can be easily reduced to checking trace inclusion on the suspension automaton.

A **test case** *t* is an LTS $< S, L_I \cup L_U \cup \{\delta\}, T, s0 >$ such that:

- *t* is deterministic and has finite behaviour
- *S* contains the terminal states *Pass* and *Fail*, with *init(Pass)* = *init(Fail)* = $\emptyset$
- for any state $s \in S$ of the test case, $s \neq$ *Pass* or *Fail*, either $init(s) = \{a\}$ for some $a \in L_I$, or $init(s) = L_U \cup \{\delta\}$.

The class of test cases over L_U and L_I is denoted as $\mathcal{TEST}(L_U, L_I)$. A **test suite** *T* is a set of test cases: $T \subseteq \mathcal{TEST}(L_U, L_I)$. L_I and L_U refer to inputs and outputs from the point of view of the IUT, so L_I represents the outputs and L_U the inputs of test cases.

The following test generation algorithm is based on the suspension automaton obtained from an LTS. It is a slightly modified version of the one in [24] which generates tests according to various implementation relations. The following one is tailored for the *ioconf* relation.

Test Generation Algorithm: Let Γ be the suspension automaton of an LTS *s*, and let $\mathcal{F}$ = *traces(s)*. Then a test case $t \in \mathcal{TEST}(L_U, L_I)$ is obtained by finite, recursive application of one of the following three non-deterministic choices:

Choice 1: Terminate the test case: $t := Pass$.

Choice 2: Give a further input to the implementation: $t := a; t'$. Here, $a \in L_I$ such that $\mathcal{F}' = \{\sigma \in L_I^\star \mid a \cdot \sigma \in \mathcal{F}\} \neq \emptyset$. To obtain t' the algorithm is applied recursively to $\mathcal{F}'$ and Γ', which is derived from $\Gamma \xrightarrow{a} \Gamma'$.

Choice 3: Check the next outputs of the implementation:

$$\begin{array}{rcl} t & := & \sum\{x; \mathit{Fail} \mid x \in L_U \cup \{\delta\}, x \notin \mathit{out}(\Gamma)\} \\ & [] & \sum\{x; t_x \mid x \in L_U, x \in \mathit{out}(\Gamma)\} \\ & [] & \delta; \mathit{Pass} \text{ if } \delta \in \mathit{out}(\Gamma) \end{array}$$

where t_x is obtained by recursively applying the algorithm for $\{\sigma \in L_\delta^\star \mid x \cdot \sigma \in \mathcal{F}\}$, and Γ' arises from $\Gamma \xrightarrow{x} \Gamma'$.

The first choice terminates the test generation procedure. Since specifications usually have infinite behaviour, test generation has to be stopped at some point. The second choice gives the next input to the implementation. Since inputs are always enabled, this step will never result in deadlock when an input is applied to the IUT. It is therefore not possible to reach a terminal *Pass* or *Fail* state. To avoid unnecessary non-determinism during testing, only one input is applied each time. The third step checks the next output of the implementation, i.e. for each $x \in L_U \cup \{\delta\}$ it is checked if *out* $(\Gamma_i$ *after* $\sigma) \subseteq out(\Gamma_s$ *after* $\sigma)$. Here, σ is the trace which has been produced so far. Any implementation producing an output *x* that does not belong to $out(\Gamma_s)$ will result in a *Fail* terminal state, indicating a non-conforming implementation. For all other outputs *x*, the generation procedure may continue. However δ does not belong to *trace(s)* so a *Pass* terminal state results and no further sequences need be checked. This test generation algorithm guarantees sound test cases with respect to *ioconf*, and the set of all possible test cases that can be obtained is exhaustive.

3. TESTING SYNCHRONOUS CIRCUITS

3.1 SYNCHRONOUS CIRCUIT MODEL

The idea of applying the above theory to validating hardware circuits comes naturally. The DILL approach uses LOTOS to specify digital circuits, so the behaviour of circuits is expressed by an LTS. On the other hand, real hardware communicates with its environment via input and output ports. An IOLTS is a realistic model since inputs are always accepted by circuits.

In this paper, only **synchronous circuits** are considered. Synchronous circuits are also referred to as **clocked** since their operation is controlled by one or more clocks. The classical synchronous circuit model is shown in Figure 1. In this model, the combinational logic provides the primary outputs and internal outputs according to the primary inputs and internal inputs. Internal outputs are then fed into state hold components to produce the internal inputs. Changes of the internal inputs are synchronised with the clock, in other words they are changed only at a particular moment of the clock cycle (usually its transition). The internal inputs affect the state of the whole circuit.

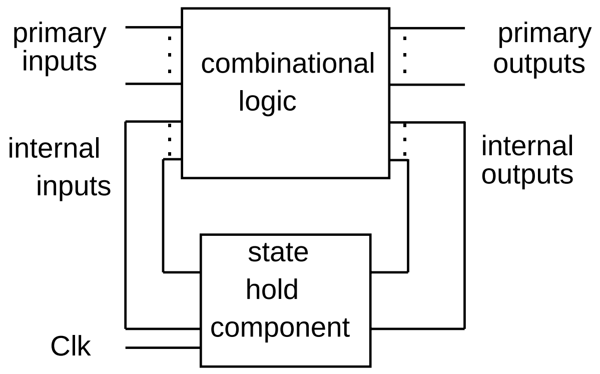

Figure 1 Synchronous Circuit Model

Because only circuit behaviour (not design) is specified in LOTOS for testing purposes, this paper addresses only behavioural modelling of synchronous circuits. Other issues such as structural modelling are discussed elsewhere [16]. A clock signal can be specified explicitly or implicitly according to convenience. LOTOS events represent signal levels during a clock cycle. Circuit behaviour is specified with reference to a clock signal. After an active clock transition, the primary outputs and internal outputs are updated according to the primary inputs and internal inputs. The rest of this section gives two illustrative examples.

A JK flip-flop is a single-bit memory element with control inputs *J* and *K*. If they are both set to 0, the flip-flop stays in the same state. If they are both set to 1, the flip-flop inverts its current value. If *J* and *K* are set to different values, the value of *J* is stored. The output is conventionally called *Q*, while its complement is *NQ* (not *Q*). The JK flip-flop specification below fixes the order in which inputs and outputs occur. This might not be a restriction of real hardware. However the order does not influence the functionality of the flip-flop, so there is no need to distinguish which input or output happens first. By restricting the event order, the state space can be substantially reduced when a component has multiple inputs and/or outputs.

```
    behaviour JK [J, K, Q, NQ] (0)                          (* initial state is 0 *)
  where
      process JK [J, K, Q, NQ] (dtQ : Bit) : noexit :=
        J ?newJ : Bit; K ?newK : Bit;                         (* get new J and K *)
        ( [(newJ eq 0) and (newK eq 0)] ->                  (* both 0 - same state *)
              Q !dtQ; NQ !not(dtQ);                       (* output current values *)
              JK [J, K, Q, NQ] (dtQ)
        []
          [(newJ eq 1) and (newK eq 1)] ->                  (* both 1 - flip state *)
              Q !not (dtQ); NQ !dtQ;                             (* invert outputs *)
              JK [J, K, Q, NQ] (not (dtQ))
        []
          [newJ ne newK] ->                                (* both differ - take J *)
          Q !newJ; NQ !not (newJ);                              (* use J as input *)
          JK [J, K, Q, NQ] (newJ) )
      endproc (* JK *)
```

The Single Pulser [22] is a standard hardware verification benchmark. It is a clocked sequential device with a one-bit input and a one-bit output. It outputs a one-cycle pulse when there is a pulse on its input. The Single Pulser can thus be used to debounce a switch, for example. Two kinds of implementations are allowed. The output pulse may be asserted on the positive-going ⌈ or negative-going ⌋ input transition, so the specification is non-deterministic. Test generation for the Single Pulser will be covered in Section 4.1. This example is introduced now to illustrate the issues in modelling synchronous circuits.

```
    process SP [Ip, Op] : noexit :=                               (* Single Pulser *)
      i; SP_P [Ip, Op] (0)                          (* +ve triggered implementation *)
```

```
[]
  i; SP_N [Ip, Op] (0)                                   (*-ve triggered implementation *)
where
  process SP_P [Ip, Op] (dtI: Bit) : noexit :=
    Ip ?newI : Bit;                                                    (* get new input *)
    ( Op !1 [(dtI eq 0) and (newI eq 1)];                   (* output 1 on 0->1 input *)
      SP_P [Ip, Op] (newI)
    []
      Op !0 [not ((dtI eq 0) and (newI eq 1))];                      (* else output 0 *)
      SP_P [Ip, Op] (newI) )
  endproc (* SP_P *)
  process SP_N [Ip, Op] (dtI: Bit) : noexit :=
    Ip ?newI : Bit;                                                    (* get new input *)
    ( Op !1 [(dtI eq 1) and (newI eq 0)];                   (* output 1 on 1->0 input *)
      SP_N [Ip, Op] (newI)
    []
      Op !0 [not ((dtI eq 1) and (newI eq 0))];                      (* else output 0 *)
      SP_N [Ip, Op] (newI) )
  endproc (* SP_N *)
endproc (* SP *)
```

The LTSs that are observationally equivalent to the above LOTOS specifications appear in figure 2. Observational equivalence is used here since conformance testing relates only to external behaviour of circuits. The equivalence is preserves all external behaviour and has much smaller state space compared to the original specifications. Figure 3 shows suspension automata built from the LTSs. Self-loops in this figure denote δ (quiescent state) actions. Figure 4 presents several tests generated from the automata using the algorithm explained in the preceding section.

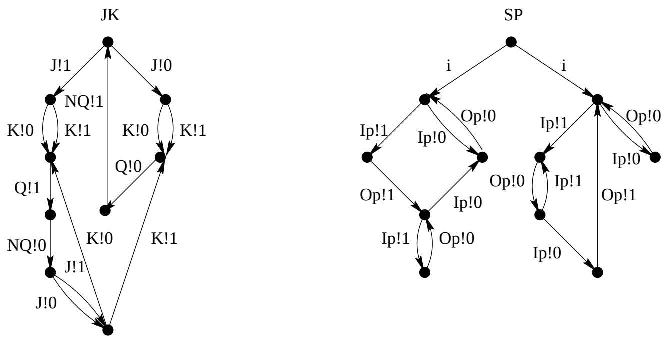

Figure 2 LTSs for the JK Flip-Flop and Single Pulser

The modelling approach discussed above has some implications for testing. Firstly, LOTOS events represents stable signal values in a specific clock cycle. When testing a circuit, applying inputs and observing outputs should also be conducted when the circuit is stable. This is not a problem for clocked circuits

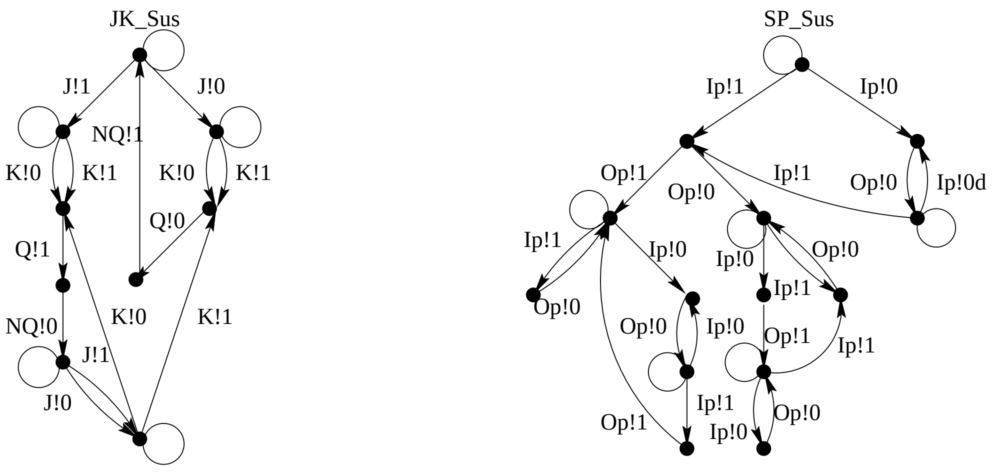

Figure 3 Suspension Automata for the JK Flip-Flop and Single Pulser

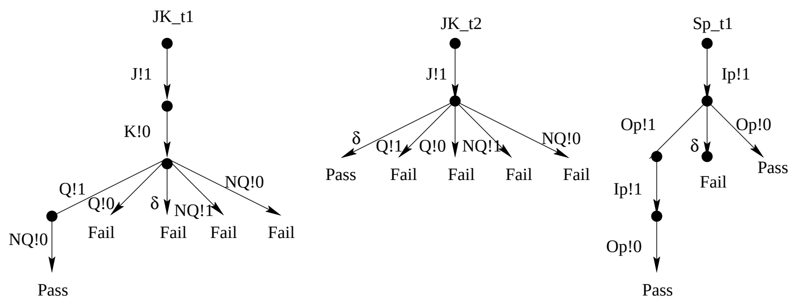

Figure 4 Several Tests of the JK Flip-Flop and Single Pulser

since the clock cycle is always chosen such that the circuit has enough time to settle down. Secondly, as stable values of inputs and outputs should appear once in every clock cycle, there is no need to worry about the δ action which indicates the absence of outputs. It is therefore less interesting to generate tests cases like *JK_t2* that check absence of outputs. They are therefore excluded from the outputs of the test generator. Finally, as discussed earlier the order of inputs and outputs is fixed to restrict the state space. In test case *JK_t1*, the test gives a *Fail* verdict when the first *NQ !0* is observed. This would not have happened if the full state space had been generated. The way to solve this problem is discussed in Section 3.2.

These two examples also indicate why the *ioconf* relation is a suitable implementation relation for validating synchronous circuits. If a specification is deterministic then *ioconf* requires that, for all possible input sequences, all the outputs of an implementation should agree with those given by the specification. This is strong enough to distinguish erroneous implementations from correct ones. On the other hand, it also permits non-deterministic specifications to be tested. For the example of the Single Pulser, a correct implementation may produce the output pulse after a ⌠ or ⎱ input. This can be properly captured by the *ioconf* relation. For example if input is initially presumed to be 0 and then it changes to 1, the output of a ⌠ implementation should be 1 (or 0 for a ⎱ implementation). As seen in test case *Sp_t1* of Figure 4, both design decisions can pass the test so implementation freedom is respected.

3.2 TEST GENERATION AND EXECUTION

The test cases generated from the algorithm in Section 2.2 have the form of a tree. This might have a straightforward mapping to TTCN (Tree and Tabular Combined Notation [13]). However, the work presented here focuses on investigating the idea of applying protocol testing theory to hardware validation. In this context it is preferable to have test cases of a simpler form that eases test execution and analysis.

A natural way to think about test cases for synchronous circuits is: for the given inputs, what should the outputs be? This indicates that test cases can be in the form of trace with alternation of inputs and outputs. For example, test case *JK_t1* can be stored in a file of the form: *J!1; K!0; Q!1; NQ!0; Pass*. In other words, transitions leading to the *Fail* verdict are not explicitly recorded. When implementations have outputs different from the one defined in the test case, a *Fail* verdict should be generated automatically. Moreover, the side-effect of not recording sequences leading to *Fail* easily solves the problem resulting from fixing the order of outputs in specifications.

This method works well with deterministic specifications. However when the specification has non-deterministic behaviour, simply generating traces from a tree raises problems. For example, the test tree of *Sp_t1* cannot be rewritten as *Ip!1; Op!1; Ip!1; Op!0; Pass* and *Ip!1; Op!0; Pass*. If a ⌠ implementation were tested by the first case, it would be given a *Fail* verdict. Conversely, a ⎱ implementation would fail the second test. Actually, both of them might be correct implementations. The problem is that an implementation has to pass all the test cases in a test suite before it is regarded as correct. But for this example, passing only one of the test cases is necessary. This is solved by marking outputs at a contradictory branch to indicate that the corresponding test is inconclusive when the marked outputs are not matched by the IUT.

At some node of a suspension automaton, suppose the test generation program finds that there are two possible output transitions with the same gate offering different values. Both of the outputs should be marked when the corresponding sequences are generated, meaning they are not necessarily matched by the implementation. Coming back to the example above, the tests then become *Ip!1; Op!1*$\star$*; Ip!1; Op!0; Pass* and *Ip!1; Op!0*$\star$*, Pass*. When output *Op!1* from the implementation is compared to the second test case, the $\star$ means this output does not have to be matched and other test outputs should be checked. If this output is matched by another test branch, comparison continues to determine if the subsequent behaviour is satisfied. As digital signals are strictly binary in the DILL model, if test generation produces both traces then no erroneous behaviours of an implementation will be missed.

Test generation is mainly based on traversing suspension automata. If Choice 1 is made, test case generation is complete and a new test case can be begun. Appending an input action to a trace corresponds to selecting Choice 2 in the test generation algorithm. Appending an output event, possibly with a $\star$ mark, equates to Choice 3.

As specifications usually have infinite behaviour, especially if they involve iterations, a test case can hardly be a complete trace unless the circuit has a deadlock state. Therefore a test suite can never cover all the behaviour of a specification. How to generate a test suite with good coverage is an important theme for testing theory based on LTSs, and is expected to be addressed at a later stage of the work presented here. In this paper, when to terminate a test case and test suite selection are mainly based on heuristics.

If covering all behaviour is not achievable, then covering all transitions might be a second-best choice. A suspension automaton is a directed graph. Generating a sequence that visits every edge in the graph at least once is the **Chinese postman problem** [5] that generates a **transition tour**. A single transition tour exists only for a strongly connected graph in which every node in the graph has a path to every other node. Otherwise, more than one tour is needed to cover all the transitions. As suspension automata may not be strongly connected, it is not possible to make direct use of transition tour generation algorithms (e.g. [9]) that guarantee the shortest tour for a strongly connected graphs. In the work presented here, the approach of [8] is adopted because it is suitable for all kinds of directed graph. In this method, depth-first search (DFS) is used whenever possible as it naturally records the transitions traversed. When an unvisited edge cannot by reached by by DFS, breadth-first search (BFS) is exploited to find a state that has an unvisited edge; DFS then continues from this state. The whole procedure repeats until this is no unvisited edge in the graph.

The CADP toolset supports an application programming interface that allows user-written programs to manipulate the state space of a given LOTOS specification. The programming interface is used to apply the test generation algorithm to synchronous circuits. For example, Figure 5 shows a test case that is generated for the JK Flip-Flop. Note that the test cases are influenced by the order in which suspension automaton edges are stored. This order is adjustable by changing parameters passed to CADP. If more coverage is required, the test generator can be re-run by using different parameters for more test cases.

Cycle	**J**	**K**	**Q**	**NQ**	**Cycle**	**J**	**K**	**Q**	**NQ**
1	1	1	1	0	2	1	1	0	1
3	0	1	0	1	4	1	0	1	0
5	0	0	1	0	6	1	1	0	1
7	0	0	0	1					

Figure 5 Part of the Test Suite Generated for the JK Flip-Flop

Each tour generated in this way is a test case and is saved in a test file. The accumulated test cases are passed to a VHDL simulator that handles a lower-level implementation of the circuit. A VHDL testbench is designed to allows the test cases to be applied and executed against the VHDL description of the circuit. The testbench mainly consists of two processes that are executed concurrently. The first process generates clock signals for the circuit under test. The second process reads the test suite file and generates signal stimuli according to the inputs of each test case. It also compares the outputs generated by the VHDL simulator with the output values specified by test case, giving a *Fail* verdict and aborting the simulation if they are not the same. The testbench also has to determine when to apply the input stimuli and to check the output result. This needs some knowledge of the circuit realisation, such as the propagation delays of components in the circuit. Special care should be given to those outputs which are marked $\star$ as discussed earlier. Between two test cases, a reset signal is generated by the testbench to re-initialise the circuit under test. The assumption is made that a circuit can always be correctly reset. The LOTOS specifications discussed previously do not specify reset behaviour, so a test need not be generated to ensure that reset is correctly achieved.

There is a peculiar situation for which the test generation program is not so suitable, namely when a circuit implementation may have non-deterministic behaviour. For such specifications, most test cases become inconclusive for many correct and incorrect implementations. This makes the test suite less meaningful. Fortunately, this is not so problematic since digital circuits are rarely allowed to be non-deterministic. Although the Single Pulser specification is non-deterministic, both of its implementations have deterministic behaviour.

4. CASE STUDIES

In this section, test generation and execution are applied to two standard benchmark circuits. [22] gives informal descriptions of these, along with circuit diagrams and timing diagrams. The Single Pulser has already been introduced. The Black-Jack Dealer is a device that plays the dealer's hand of the card game also known as Pontoon or Vingt-et-Un. Although the Single Pulser circuit is relatively small, the Black-Jack Dealer has significant complexity.

4.1 SINGLE PULSER

The formal specification of the Single Pulser was given in Section 3.1, along with its automata forms in Figures 2 and 3. The deterministic design given in the benchmark documentation issues an output after a positive transition of the input. The test generation program produces test cases such as:

Test Case 1: *Ip!0; Op!0; Ip!0, Op!0; Ip!1; Op!0⋆; Ip!0; Op!1; Ip!0; Op!0; Ip!1; Op!0; Ip!1; Op!0; Pass*

Test Case 2: *Ip!1; Op!1⋆; Ip!0; Op!0; Ip!0; Op!0; Ip!1; Op!1; Ip!1; Op!0; Pass*

When the VHDL testbench is used to execute the first test case, simulation is interrupted after the third clock cycle when an input of 1 has been supplied. The circuit outputs 1 but the expected test output is 0. As this output is marked with ⋆, the simulator concludes that the test is inconclusive for this design. The second test, however, successfully passes the simulation, indicating that the implementation can be regarded as correct. Note that the single pulser does not have a reset input, so simulation has to be restarted when the circuit is checked against the second test case.

4.2 BLACK-JACK DEALER

The Black-Jack dealer inputs are *Card_Ready* and *Card_Value* (Ace..King, Clubs..Spades). Its outputs have boolean values: *Hit* (card needed), *Stand* (stay with current cards) and *Broke* (total exceeds 21). The *Card_Ready* and *Hit* signals are used for a handshake with a human operator. Aces have value 1 or 11 at the choice of the player. Numbered cards have values from 2 to 10. Jack, Queen and King count as 10. The Black-Jack dealer is repeatedly presented with cards. It must assert *Stand* (when its score is 17 to 21) or *Broke* (when its score exceeds 21). In either case the next card starts a new game.

In the LOTOS specification of the Black_Jack dealer, a new data type *Value* is defined to represent the card value. Although the LOTOS standard data type *NaturalNumber* might appear suitable, CADP cannot generate the corresponding LTS for an infinite data type like this. The key point in the specification is how to handle the ambiguous value of an Ace. To solve the problem, the specification uses the method given by [27]. Specification behaviour occupies about 80 lines including comments. (The circuit diagram is also about a page.)

Using CADP and the test generator program written by the authors, a test suite for the Black-Jack Dealer was derived. The test suite is able to test 181 different hands of cards that a dealer may hold. The VHDL implementation given in [27] was evaluated against this test suite.

Although the circuit was expected to pass the test suite, a *Fail* verdict was recorded after the dealer was given the following cards: 5, 5, 3, 2, 1, 10. In this case the dealer should be *Broke* because the sum of the cards is 26. However the circuit outputs neither *Stand* nor *Broke* since it considers the total to be just 16. Other card combinations including an Ace and causing *Broke* exhibited the same problem. This showed the problem was related to processing an Ace.

The circuit should initially take an Ace as 11. It should be re-valued as 1 (subtracting 10 from the sum) the first time the result would be *Broke*. If the following cards would make the sum exceed 21, no re-valuation should be done as no Ace is 11. But the given benchmark design still re-values the Ace card, so the circuit is not *Broke* in this case. Carefully simulating the circuit discovered a problem with one of the flag registers (*Ace11Flag* in [22]). This indicates if there is an Ace to be 11. It is not reset to zero properly because the effective duration of the signal used to reset it (*ClearAce11Flag*) is too short. By slightly modifying the circuit to remove the cause of this short duration, the circuit was able to successfully pass the test suite.

5. CONCLUSION AND FUTURE WORK

The framework of formal methods in protocol testing has been used for testing digital circuits. The protocol testing implementation relation *ioconf* has been justified as suitable for testing synchronous hardware circuits. A prototype tool has been developed to generate and execute test cases automatically. The approach has been validated on standard hardware verification benchmarks. It is noteworthy that it could find an error in a published circuit design for the Black-Jack Dealer.

Future work will include applying test selection techniques while generating test cases. Test cases are guaranteed to cover all the transitions of the specification state space, but no further coverage information can be provided. To progress further will involve defining a suitable coverage function or exploiting some existing selection methodologies (e.g. [1, 3]). Applying the method presented in this paper to higher-level specification of digital circuits will also be interesting. In the examples of this paper, specification is relatively close to real implementation. For example, signals are in one-to-one correspondence between both levels. This makes the task of mapping the test cases to the actual implementation very easy. But when specification is more abstract, greater effort will be needed to make this correspondence.

References

[1] J. Alilovic-Curgus and S. T. Vuong. A metric based theory of test selection

and coverage. In A. A. S. Danthine, G. Leduc, and P. Wolper, editors, *Proc. Protocol Specification, Testing and Verification XIII*, pages 289–304. North-Holland, 1993.

[2] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. K. Sabnani, editors, *Proc. Protocol Specification, Testing and Verification VIII*. North-Holland, Amsterdam, Netherlands, June 1988.

[3] O. Charles and R. Groz. Basing test coverage on a formalization of test hypotheses. In M. Kim, S. Kang, and K. Hong, editors, *International Workshop on Testing Communicating Systems X*, pages 109–124. Chapman-Hall, London, UK, 1997.

[4] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theory of Computer Science*, pages 83–133, 1984.

[5] J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1972.

[6] 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.

[7] J. C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, Berlin, Germany, 1996.

[8] 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.

[9] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

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

[11] IEEE. *IEEE Standard Hardware Design Language based on the Verilog Hardware Description Language*. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.

[12] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.

[13] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Conformance Testing Methodology and Framework – Part 3: The Tree and Tabular Combined Notation (TTCN)*. ISO/IEC 9646-3. International Organization for Standardization, Geneva, Switzerland, 1991.

[14] Ji He and K. J. Turner. Extended DILL: Digital logic with LOTOS. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, Nov. 1997.

[15] Ji He and K. J. Turner. Timed DILL: Digital logic with LOTOS. Technical Report CSM-145, Department of Computing Science and Mathematics, University of Stirling, UK, Apr. 1998.

[16] Ji He and K. J. Turner. Modelling and verifying synchronous circuits in DILL. Technical Report CSM-152, Department of Computing Science and Mathematics, University of Stirling, UK, Feb. 1999.

[17] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, Aug. 1992.

[18] D. Moundanos, A. Abraham, and Y. V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47:2–14, 1998.

[19] R. D. Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[20] D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, Dec. 1990.

[21] 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.

[22] J. Staunstrup and T. Kropf. IFIP WG10.5 benchmark circuits. http://goethe.ira.uka.de/hvg/benchmarks.html, July 1996.

[23] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:25–59, 1996.

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

[25] K. J. Turner and R. O. Sinnott. DILL: Specifying digital logic in LOTOS. In R. L. Tenney, P. D. Amer, and M. Ü. Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86. North-Holland, Amsterdam, Netherlands, 1994.

[26] F. Vemuri and R. Kalyanaraman. Generation of design verification tests from behavioral VHDL programs using path enumeration and constraint

programming. *IEEE Transactions on Very Large Scale Integration Systems*, 3:201–214, 1995.

[27] D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1980.