Kenneth J. Turner and Ji He. Formally-Based Design Evaluation (extended version of article published by and copyright Springer-Verlag). In Tiziana Margaria and Thomas F. Melham, editors, Proc. 11th Conference on Correct Hardware Design and Verification Methods (CHARME 2001), Lecture Notes in Computer Science 2144, pages 104-109, Springer Verlag, Berlin, Germany, September 2001.

Formally-Based Design Evaluation

Kenneth J. Turner and Ji He

Computing Science and Mathematics, University of Stirling, Stirling FK9 4PU, Scotland
kjt@cs.stir.ac.uk, h.ji@reading.ac.uk

Abstract. This paper investigates specification, verification and test generation for synchronous and asynchronous circuits. The approach is called DILL (Digital Logic in LOTOS). DILL models are discussed for synchronous and asynchronous circuits. Relations for (strong) conformance are defined for verifying a design specification against a high-level specification. An algorithm is also outlined for generating and applying implementation tests based on a specification. Tools have been developed for automated test generation and verification of conformance between an implementation and its specification. The approach is illustrated with various benchmark circuits as case studies.

Keywords: asynchronous design, conformance, LOTOS (Language Of Temporal Ordering Specification), synchronous design, test generation, verification

1 Introduction

DILL (Digital Logic in LOTOS [7–13, 19]) is an approach for specifying digital circuits using LOTOS (Language Of Temporal Ordering Specification [6]). DILL has been developed over six years to allow formal specification of hardware designs, represented using LOTOS at various levels of abstraction. DILL addresses functional and timing aspects, synchronous and asynchronous design. There is support from a library of common components and circuit designs. Analysis uses standard LOTOS tools.

LOTOS is a formal language standardised for use with communications systems. DILL, which is realised through translation to LOTOS, is a substantially different application area for this language. Since the readers of this paper are unlikely to be familiar with LOTOS or its communications origins, much of the technical detail has been omitted. However the articles on DILL referenced above can provide additional background.

LOTOS can be used to support rigorous specification and analysis of hardware. LOTOS is neutral with respect to whether a specification is to be realised in hardware or software, allowing hardware-software co-design. LOTOS inherits a well-developed verification theory from the field of process algebra, and has a theory for testing and test generation. There is good support from general-purpose LOTOS toolsets such as CADP (Cæsar Aldébaran Development Package [3]).

Languages such as VHDL (VHSIC Hardware Description Language), Verilog and ELLA are commonly used in industry. These languages are semi-formal because their semantics is based on simulation models. Other languages do have formal semantics, e.g. CIRCAL (Circuit Calculus), HOL (Higher Order Logic) and Ruby. DILL most closely resembles CIRCAL in that both have a behavioural basis in process algebra. At a low level of specification, the true concurrency semantics of CIRCAL are perhaps more appropriate than the interleaving semantics of LOTOS. However, the integrated

data typing in LOTOS makes it much more expressive than CIRCAL. In the authors' experience, DILL can be used successfully at a variety of abstraction levels where CIRCAL appears to be less effective.

The present paper synthesises several aspects of DILL for modelling and verifying synchronous and asynchronous circuits. Synchronous design is important as it is the main approach for digital technology. Control by clock signals makes it easier to abstract away from timing information. The current standard for LOTOS does not support quantified timing, although the authors have developed Timed DILL [10] for hardware timing analysis using ET-LOTOS (Enhanced Timed LOTOS [15]). However, asynchronous circuits are also interesting as they operate at the speed of individual components. Unfortunately asynchronous circuits are much harder to design and to analyse.

Much of the early work on hardware verification used theorem-proving. The trend is now to combine theorem-proving and model-based approaches so as to achieve generality as well as automated support. LOTOS verification approaches tend to be state-based using an LTS (Labelled Transition System). Current LOTOS tools offer model checking and reachability analysis, together with equivalence or preorder checking. DILL can thus exploit a range of verification techniques.

Verification of asynchronous circuits, especially those that are speed-independent or delay-insensitive, has been a research topic for some years [1, 2]. Rigorous testing of asynchronous circuits is still in its infancy. Like DILL, 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* [1], *decomposition* [2] and *strong conformance* [4]. 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 DILL, tests are derived from higher-level specifications in a novel adaptation of protocol conformance testing theory.

2 Modelling Approach

DILL supports logic designs at different levels of abstraction, with formal comparison of higher level and more detailed design specifications. DILL does not provide guidelines for refinement, since design is presumed to follow conventional engineering practice. DILL supports behavioural as well as structural specification.

Wires or tracks between components are not normally represented explicitly in DILL. A component's ports (e.g. its pins) are represented by LOTOS event gates. To 'wire up' two ports, their LOTOS gates are merely synchronised. Since LOTOS allows multi-way synchronisation, it is easy to connect one output to several inputs. In high-speed circuits, the transmission time over a wire may be modelled as a delay. It may also be necessary to model wires in asynchronous designs.

2.1 Synchronous Circuits

LOTOS, like most specification languages, deals only with discrete events. In synchronous circuits, changes in signal level are controlled by clock pulses (except for components such as level-triggered flip-flops). Signal levels can thus be treated as maintained during a clock cycle, i.e. one LOTOS event per clock cycle.

The classical synchronous circuit model has combinational logic to provide the primary outputs and the internal outputs according to the primary inputs and the 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 primary inputs are usually synchronised with the clock signal. This makes designing and analysing synchronous circuits much easier. DILL incorporates this practice into its synchronous circuit model, assuming that the primary inputs have already been synchronised with the clock signal.

For a synchronous circuit, designers must ensure that the clock cycle is slower than the slowest stage in a circuit. This can be done by analysing the timing characteristics of components used in the circuit. The untimed version of DILL cannot of course confirm if the clock constraint is met. However as discussed elsewhere [10], Timed DILL can specify such constraints. Properly modelling the storage components and environment can ensure that the clock constraint is always fulfilled by a DILL specification.

The fundamental DILL model always allows an input or output port to offer an event corresponding to a signal change. This model is generally applicable, but may lead to non-determinism due to the lack of quantified timing [11]. The model therefore has to be constrained according to the environment in which it operates. If the clock is slow enough to let every signal settle down, it is reasonable to allow the value of each signal to change just once per clock cycle.

The DILL synchronous model imposes further two restrictions. It is important that there is no cyclic connection within a stage, and storage components have to be specified in the behavioural style. These restrictions are related to the component model, for otherwise a DILL specification might deadlock where a real circuit could still work.

2.2 Asynchronous Circuits

In asynchronous DILL, it is only signal changes that are modelled. 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 methods handle delay-insensitive, quasi delay-insensitive or speed-independent circuits.

Bounded delays need a formalism that can specify quantitative timing. DILL deals with (quasi) delay-insensitive and speed-independent circuits since they assume unbounded delays that are appropriate for LOTOS. (Quasi) delay-insensitive designs can be easily changed to speed-independent circuits by inserting artificial delay components. Most wire delays can in fact be absorbed into the preceding components. Only components with more than one output need special treatment.

Asynchronous DILL concentrates mainly on speed-independent design. Happily this is 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. If new inputs cannot change any pending outputs, the design is termed semi-modular. Semi-modularity is often used as a correctness criterion for speed independence, since the violation of semi-modularity causes speed-dependent behaviour.

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 LOTOS 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. 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 first determinised (internal non-determinism is removed). Outgoing edges are then 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.

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, making verification more difficult.

Assumptions about the environment of an asynchronous circuit 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]. This assumes that the environment does not provide extra inputs, so inputs that are accepted only by an 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 used when verifying asynchronous circuits. Relations are defined that respect 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 Conformance Testing and Verification

Conformance testing is a term drawn from communications systems to mean evaluating the correctness of an implementation against its specification. 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 (Labelled Transition Systems), both the specification and the IUT (Implementation Under Test) are modelled as LTSs. An IUT communicates with its environment through symmetric interactions, so its environment as expressed by tests is also modelled as an LTS.

Many real-world systems communicate with their environment in a different way from an LTS, with a clear distinction between inputs and outputs. 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 [18] this kind of behaviour is modelled as an IOLTS (Input-Output Labelled Transition System), which is a special kind of LTS. In an IOLTS, the set of actions is partitioned into inputs and outputs. Intuitively this means that input actions are always enabled in any state.

The DILL approach allows conformance of an implementation to be verified against its specification. More precisely, a design specification is formally checked against an abstract specification. The same approach also allows test suites for an implementation to be rigorously derived from its specification. This allows the actual implementation to be checked for correctness. Since implementations are not usually formally derived from specifications, rigorously derived tests are an important solution to formal gaps in the design process.

3.1 Conformance Testing

In the DILL approach to conformance testing, a circuit is specified in LOTOS (whose semantics is given by an LTS). The implementation of the same circuit is described by VHDL. The behaviour of a VHDL program is presumed to be modelled by an IOLTS that need not be known explicitly.

To support the checking of conformance, an intermediate LTS termed a suspension automaton is built from the specification LTS. The suspension automaton of an LTS

is obtained by adding self-loops for all quiescent states (where no output is pending). The resulting automaton is then determinised. The important properties of a suspension automaton are that it is deterministic and that it respects the outputs of the original LTS. Checking conformance can be easily reduced to checking trace inclusion on the suspension automaton.

A test case is an LTS with finite deterministic behaviour. A test case ends with states labelled *Pass* or *Fail* to indicate the verdict of conformance. The test cases generated by DILL 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 [5] 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.

Tests are generated from a suspension automaton by an algorithm that offers three choices in each iteration. The first choice terminates test generation. 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. 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 choice checks each possible next output of the implementation. Any implementation producing an unexpected output will result in a *Fail* terminal state, indicating a non-conforming implementation. For all other outputs, test generation may continue. This test generation algorithm guarantees sound test cases for *ioconf* (input-output conformance). An exhaustive set of test cases is also guaranteed.

This method works well with deterministic specifications. However when the specification has non-deterministic behaviour, simply generating traces from a tree raises problems. The problem is that an implementation has to pass all the test cases in a test suite before it is regarded as correct. This problem is solved by marking outputs at a contradictory branch to indicate that the corresponding test is inconclusive when the marked output is not matched during testing.

The CADP toolset for LOTOS supports an application programming interface that allows user-written programs to manipulate the state space of a given LOTOS specification. The authors have developed the *TestGen* tool to realise the test generation algorithm. Each generated transition tour 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 was designed to allow 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 cases. A *Fail* verdict is given and the simulation is aborted if these outputs are not the same. This needs some knowledge of the circuit realisation, such as the propagation delays of components in the circuit. Between two test cases of a test suite, a reset signal is generated by the testbench to re-initialise the circuit under test. (It is assumed that a circuit can always be reset.)

An inconclusive point may be met during testing because there are alternative implementations. In such a case the testbench may have to search forward or backward to find a matching alternative. This means that loops can arise during testing, so the testbench operates a strategy to avoid endless repetition. The testbench also needs to maintain a timer. If there is no output within a certain period, the circuit must be assumed quiescent (waiting for input). A failure verdict must be given if output was required at this point.

3.2 Conformance Verification

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 for writing 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* (input-output conformance), several other definitions have to be introduced. A quiescent state is one that cannot perform any output transitions or an internal transition. An implementation has input-output conformance to its specification if, after every trace 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 a quiescent state, the implementation may not output if the specification cannot do so. As for test generation, a suspension automaton is created as a step towards verifying conformance.

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 implementation relation *confor* (conformance), 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*.

If *sp* can produce output *op*, 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*. 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 overcome this problem, a special 'action' δ is introduced for quiescence, meaning 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.

The *confor* relation requires that, after a suspension trace of *Spec*, the outputs that an implementation *Impl* can produce are included in what *Spec* can produce. If *Impl* can follow the suspension trace, the inputs that *Spec* can accept are also accepted by *Impl*. A second implementation relation *strongconfor* (strong conformance) is also defined. This is similar 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.

To check these relations, a specification LTS is first transformed into a suspension automaton. This is part of the verification tool *VeriConf* developed by the authors to check the *(strong)confor* relations. 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.

4 Case Studies

This section illustrates the DILL approach to conformance testing and verification using various benchmark circuits. By choosing examples defined by others, the authors have avoided biasing the illustrations in favour of their method. A selection has been made from case studies by the authors to illustrate larger and smaller circuits, synchronous and asynchronous designs, testing and verification.

4.1 Bus Arbiter

The Bus Arbiter is a benchmark circuit using to check hardware verifiers [17]. It is a synchronous circuit that grants access on each clock cycle to a single client among a number of clients requesting use of a bus. The inputs to the arbiter are a set of request signals from clients. The outputs are a set of acknowledge signals that indicate which client is granted access during a clock cycle. For brevity the specifications and formal analysis are not given here, but can be found in [11, 13].

The arbitration algorithm embodied in the design is a round-robin token scheme with priority override. Normally the arbiter grants access to the highest priority client: the one with the lowest index number among all the requesting clients. However as requests become more frequent, the arbiter is designed to fall back on a round-robin scheme so that every requester is eventually acknowledged. This is done by circulating a token in a ring of arbiter cells, with one cell per client. The token moves once every

clock cycle. If a client's request persists for the time it takes for the token to make a complete circuit, that client is granted immediate access to the bus.

LOTOS supports specification at various levels of abstraction. Although the Bus Arbiter has been studied by many researchers, as far as the authors know there has not been a formal specification of the arbitration algorithm used in the design. With LOTOS, it is possible to provide such a higher-level specification. Translating the algorithm to LOTOS is quite straightforward. For an arbiter with three cells, the LOTOS specification has 79 lines (including comments) for the behavioural specification.

The design of the arbiter consists of repeated cells. Each cell is in charge of accepting request signals from a client, and sending back acknowledgements to the same client. The first cell is slightly different because it is assumed that the token is initially in the first cell. Because the components of each cell are in the DILL library, it is very easy to specify the process describing a cell. The specification of an arbiter with three cells is obtained by connecting three such processes. There is also an environment constraint in the structural specification of the arbiter to meet the conditions of the synchronous circuit model (section 2.1).

The formulation of properties with the CADP tool uses action-based temporal logic, namely ACTL (Action-based Computational Tree Logic [16]) and HML (Hennessy-Milner Logic). The following three properties have to be proved for the circuit: no two acknowledge outputs are asserted in the same clock cycle (safety); every persistent request is eventually acknowledged (liveness); and acknowledge is not asserted without request (safety). To verify the higher-level specification against the temporal logic formulae, the LTS of the specification was produced first. CADP generates an LTS with 3649 states and 7918 transitions. This is reduced to 379 states and 828 transitions with respect to strong bisimulation. Both generation and reduction take a few seconds on a 300 MHz Sun. The temporal logic formulae are then verified against the minimised LTS within a minute.

The real challenge comes when the lower-level design specification is verified. The state space is so large that direct generation of the LTS from the LOTOS specification is impractical. Compositional generation was therefore used to verify the arbiter. The specification is divided into several smaller specifications to make sure that it is possible to generate an LTS for each of them. Then these are reduced with respect to a suitable equivalence relation.

In order to get valid verification results, special attention must be given to the equivalence relation that is used. Safety equivalence preserves all safety properties, while branching bisimulation equivalence preserves liveness properties when there are no livelocks in specifications. Both of these equivalences are congruences with respect to the LOTOS parallel and hide operators. These two equivalences are thus appropriate to compositional generation.

The design of the arbiter was divided into three pieces, one per cell of the arbiter. An LTS which is safety equivalent to the LOTOS specification of the design was generated in about seven minutes. The two safety properties were verified to be true against this LTS, implying that the design also satisfies these properties. Verification of the formulae took just seconds. However generating an LTS that is branching equivalent to the design took almost one day, after which the liveness property was also verified to be true.

For checking equivalence between the higher-level algorithm and the lower-level design, compositional generation was exploited to generate the LTS for the design. This time each cell was reduced with respect to observational equivalence since it is a congruence for the LOTOS parallel and hide operators. The LTS was generated in about eight minutes. It was expected that this LTS would be observationally equivalent to the one representing the higher-level specification. However CADP discovered that they are not! The tool provides a counter-example in which client 0 requests the bus during the first three cycles. The high-level and low-level specifications both grant access to this client. In the fourth cycle, client 0 cancels its request but client 1 begins to request access. At this point the two levels of specifications are different: the low-level specification offers 0 for *Ack1*, whereas the high-level specification offers 1 for *Ack1*.

After step-by-step simulation of the counter-example, it was soon discovered that the circuit does not properly reset the override out signal. This is a fault in the supposedly proven benchmark circuit. The design was modified and then verified to be observationally equivalent to the higher-level algorithmic specification. The corrected circuit diagram appears in [11].

4.2 Black-Jack Dealer

The Black-Jack Dealer is another verification benchmark circuit [17], also discussed in [20]. Black-Jack is a card game also known as Pontoon or Vingt-et-Un. For brevity the specifications and formal analysis are not given here, but can be found in [11, 12].

The Black-Jack Dealer is a synchronous circuit whose inputs are *Card_Ready* and *Card_Value* (Ace..King, Clubs..Spades). Its outputs are boolean: *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 user. 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 standard LOTOS 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 [20]. Specification behaviour occupies about 80 lines including comments. (The circuit diagram is also about a page.)

Using CADP and the authors' *TestGen* program, 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 [20] 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. The same problem was found with other card combinations including an Ace that should cause *Broke*.

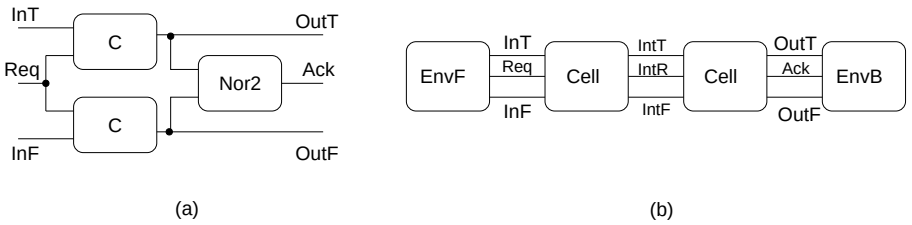

Fig. 1. Implementation of A Two-Stage FIFO from Individual Cells

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, there should be no re-valuation as no Ace is 11. But the given design still re-values the Ace card, so the circuit is not *Broke* in this case. Carefully simulating the circuit discovered a problem in the benchmark with one of the flag registers that indicates an Ace should be 11. The register is not reset to zero properly because the effective duration of the signal used to reset it is too short. By slightly modifying the circuit to remove the cause of this short duration, the circuit was able to pass the test suite successfully. Again, the authors had discovered a flaw in a supposedly verified design.

4.3 Asynchronous FIFO

As a typical asynchronous circuit, an asynchronous FIFO buffer was specified and analysed. For brevity the specifications and formal analysis are not given here, but can be found in [7, 8, 14].

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. A possible implementation for a FIFO stage is given in figure 1 (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 (transition synchronisers used in asynchronous circuits).

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. A two-stage FIFO can then be implemented as in figure 1 (b).

The specification should exhibit liveness. Using CADP, it was verified that the specification satisfies the following property: if there is an input of *1*, then the output will eventually become *1*. The property for input of *0* is similar and was also shown to be true. It was verified that *Spec* $\approx$ *Impl* $||$ *(EnvB* $||[\cdots]||$ *EnvF)*, where $\approx$ denotes observational equivalence.

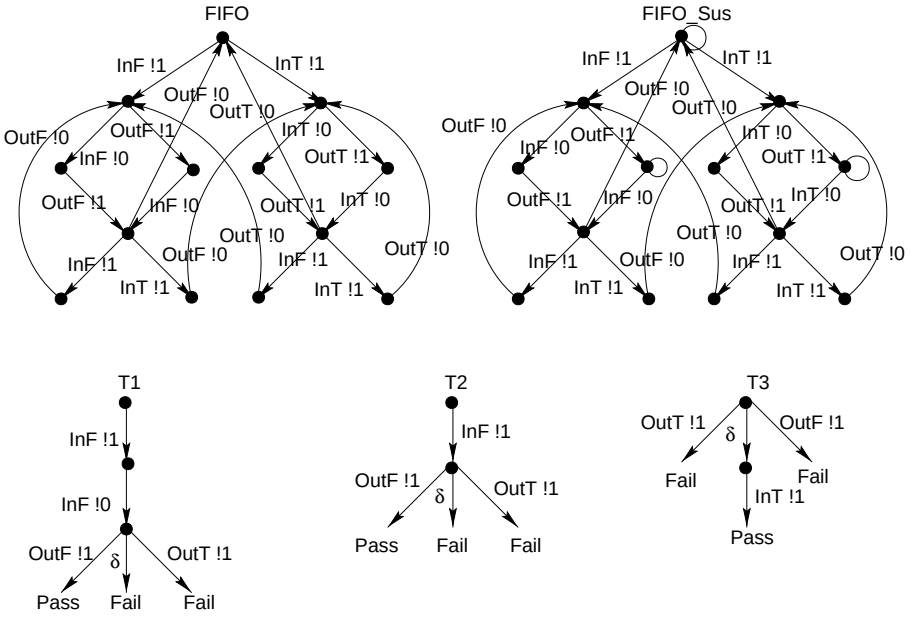

Fig. 2. LTS, Suspension Automaton and Several Tests of FIFO

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 implementation specification in quasi-receptive style. It uses the corresponding DILL library components. The corresponding quasi-receptive specifications *EnvF_QR* and *EnvB_QR* also have to be written. *EnvF* has no inputs and so is identical to *EnvF_QR*.

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 liveness property is also satisfied by the implementation *Impl_QR* || *(EnvB_QR* $||[\cdots]||$ *EnvF_QR)*.

Figure 2 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 δ (quiescent) 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* || *(EnvB_QR* $||[\cdots]||$ *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

4.4 Selector

A selector (an asynchronous design component) allows non-deterministic choice of output. For brevity the specifications and formal analysis are not given here, but can be found in [7, 8, 14].

After a change on input *Ip*, either *Op1* or *Op2* may change depending on the implementation. Figure 3 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.

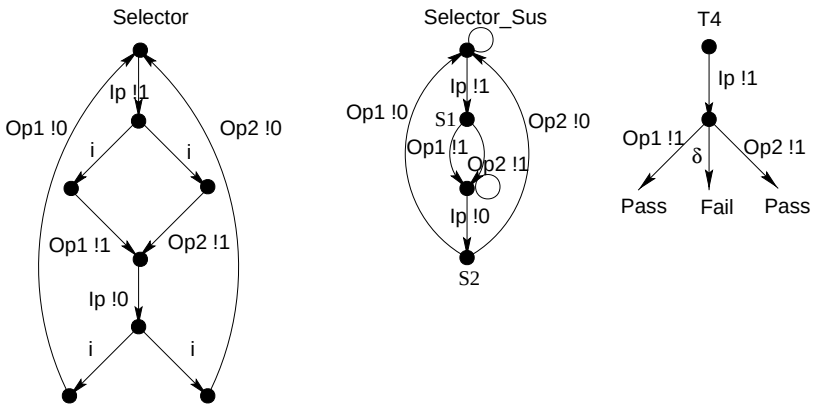

Fig. 3. LTS, Suspension Automaton and One Test of Selector

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*, the output *Op1 !1* is marked with the current state ($\star$S1) since an implementation may also do *Op2 !1*. 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 ($\star$S1)	Ip !0	Op1 !0 ($\star$S2)	δ	Ip ! 1
Op2 !1 ($\star$S1)	δ	Ip !0	Op2 !0 ($\star$S2)	Pass	

5 Conclusion

An approach to specifying synchronous circuits has been presented. This has allowed standard hardware benchmarks to be verified – the Bus Arbiter and the Black-Jack Dealer in this paper. The authors were pleasantly surprised to find that their approach discovered previously unknown flaws in these circuit designs.

An approach to specifying asynchronous circuits has also been presented. (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. The *(strong)confor* relations have been defined to assess the implementation of an asynchronous circuit against its specification.

In comparison to other techniques applied to the same case studies, e.g. COSPAN and CIRCAL, DILL is much more convenient for giving a higher-level specification. CIRCAL gives an abstract view of a synchronous circuit by directly specifying its corresponding finite state machine, but this is not always a natural representation of circuit behaviour.

Being based on process algebra, DILL specifications can be verified by equivalence and preorder checking. This is distinctive in that most hardware verification systems are based on theorem proving or model checking. Equivalence or preorder checking makes it possible to write the specification in the same formalism as the implementation, here DILL (or really, LOTOS). The correctness of a DILL specification can be easily checked by simulation tools. 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. The *VeriConf* tool was developed to support the *(strong)confor* relations.

However, the size of the circuit that can be effectively verified is small compared to that handled by other mature hardware verification tools. There are two main reasons for this performance limitation. One comes from the modelling language LOTOS, and the other comes from the CADP tool. For synchronous circuits, the order in which signals occur during a clock cycle is not so important. So it is reasonable to imagine that the inputs happen together and then output occurs. But when modelling such circuits in DILL, independent (interleaved) inputs are allowed so the state space is considerably enlarged. CADP is still under development, and most of its features are currently based on explicit state exploration. On-the-fly observational equivalence checking is not currently supported by CADP. A BDD representation of LOTOS specifications is still being developed for CADP, although BDDs are used to represent intermediate data types in some algorithms. With tool improvements, the verification performance reported in this paper can be expected to improve.

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 Digital Circuit Design*, Workshops in Computing, pages 51–103. Springer-Verlag, 1995.

3. 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.
4. 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.
5. 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.
6. 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.
7. Ji He. *Formal Specification and Analysis of Digital Hardware Circuits in LOTOS*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, UK, Apr. 2000.
8. Ji He. Formal specification and analysis of digital hardware circuits in LOTOS. Technical Report CSM-158, Department of Computing Science and Mathematics, University of Stirling, UK, Aug. 2000.
9. 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.
10. 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.
11. 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, Apr. 1999.
12. 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.
13. 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.
14. Ji He and K. J. Turner. Verifying and testing asynchronous circuits using LOTOS. In T. Bolognesi and D. Latella, editors, *Proc. Formal Methods for Distributed System Development (FORTE XIII/PSTV XX)*, pages 267–283, London, UK, Oct. 2000. Kluwer Academic Publishers.
15. L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 28:271–292, May 1996.
16. 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.
17. J. Staunstrup and T. Kropf. IFIP WG10.5 benchmark circuits. http: //goethe. ira. uka.de/hvg/ benchmarks.html, July 1996.
18. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17:103–120, 1996.
19. 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, Amsterdam, Netherlands, 1994. North-Holland.
20. D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1980.