# 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.** The paper investigates specification, verification and test generation for synchronous and asynchronous circuits. The approach is called DILL (Digital Logic in LOTOS – the ISO Language Of Temporal Ordering Specification). Relations for (strong) conformance are defined to verify a design specification against a high-level specification. Tools have been developed for automated testing and verification of conformance between an implementation and its specification.

## 1 Introduction

DILL (Digital Logic in LOTOS [5]) is an approach for specifying digital circuits using LOTOS (Language Of Temporal Ordering Specification [4]). DILL allows formal specification of hardware designs, represented using LOTOS at various levels of abstraction. DILL deals with 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. Among Hardware Description Languages, DILL most closely resembles CIRCAL (Circuit Calculus) in that both have a behavioural basis in process algebra. In the authors' experience, DILL can be used successfully at a variety of abstraction levels where CIRCAL appears to be less effective.

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. LOTOS is neutral with respect to whether a specification is to be realised in hardware or software, allowing hardware-software co-design. LOTOS has well-developed theories for verification and test generation.

The current standard for LOTOS does not support quantified timing, although the authors have developed Timed DILL for hardware timing analysis using ET-LOTOS (Enhanced Timed LOTOS). However, asynchronous circuits are also of interest. Like DILL, other asynchronous verification approaches define relations that judge correctness of a circuit design. However it is not possible to detect deadlocks and livelocks with *conformance* [1] and *decomposition* [2]. Although *strong conformance* [3] can do this, it does not work for non-deterministic specifications. The relations *confor* and *strongconfor* mentioned in this paper resolve these problems.

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 Synchronous and Asynchronous Circuit Models

DILL supports logic designs at different levels of abstraction, with formal comparison of higher level and more detailed design specifications. 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.

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. 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 DILL model is 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. DILL also requires that there be no cyclic connection within a stage, and storage components have to be specified in the behavioural style.

Some of the better-known asynchronous design methods handle delay-insensitive, quasi delay-insensitive or speed-independent circuits. 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. 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. 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.

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. 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. Internal events must first be determinised (i.e. internal non-determinism is removed). Outgoing edges are then added to create input quasi-receptive specifications. Since it can be hard to extract an input quasi-receptive environment from a behavioural specification, 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, Dill describes behaviour using an IOLTS (Input-Output Labelled Transition System) whose actions are partitioned into inputs and outputs.

Several implementation relations have been defined to express conformance of an implementation to its specification. The authors have defined the *confor* (conformance) relation to require that, after a suspension trace of the specification, the outputs that an implementation can produce are included in what the specification can produce. 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. The verification tool *VeriConf* developed by the authors checks the *(strong)confor* relations.

The *ioconf* relation (input-output conformance) has been defined to reflect the input-output relationship between an implementation and its specification. Suppose that *sp* is a state of the specification and that *im* is the corresponding state in the implementation. If *sp* can produce output *op*, a correct implementation should also produce it. If *sp* cannot produce a certain output, neither should the 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 suitable relation should thus require output inclusion instead of output equality. Unfortunately a circuit that accepts everything but outputs nothing would also be qualified as a correct implementation. The overcome this problem, a special 'action' $\delta$ is introduced for quiescence, meaning the absence of output. Like any other output, if $\delta$ 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 Dill approach allows a design specification to be formally checked against an abstract specification. The same approach also allows test suites for an implementation to be rigorously derived from its specification. A circuit is specified in Lotos (whose semantics is given by an LTS – an ordinary Labelled Transition System). The implementation of the same circuit is described by VHDL.

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 ($\delta$ 'actions' where no output is pending). The resulting automaton is then determinised. Checking conformance is reduced to checking trace inclusion on the suspension automaton. A test case has finite deterministic behaviour that ends with states labelled *Pass* or *Fail* to indicate the verdict of conformance. The test cases generated by Dill have the form of traces to allow easy measurement of test coverage and automatic execution of test cases. The strategy is to cover all transitions in a transition tour that addresses the Chinese Postman problem.

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. 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 and exhaustive test cases for the *ioconf* relation. The authors have developed the *TestGen* tool to realise this algorithm. Each generated transition tour is a test case and is saved in a test file. A testbench was designed to allow the test cases to be applied and executed against the VHDL description of the circuit.

## 4 Case Studies

### 4.1 Bus Arbiter

The Bus Arbiter is a benchmark circuit used to exercise hardware verifiers. 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. 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..

The formulation of properties uses action-based temporal logics ACTL (Action-based Computational Tree Logic [6]) 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. Generation and minimisation of the LTS take a few seconds on a 300 MHz Sun. The temporal logic formulae are then verified against the minimised LTS within a minute.

To check the lower-level specification, the design of the arbiter was divided into 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 took about eight minutes to calculate. In fact this LTS is not observationally equivalent to the one representing the higher-level specification. It was found 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.

### 4.2 Black-Jack Dealer

The Black-Jack (Pontoon, Vingt-et-Un) Dealer is another verification benchmark circuit [7]. It is a synchronous circuit whose inputs are *Card_Ready* and *Card_Value*. Its outputs are boolean: *Hit* (card needed), *Stand* (stay with current cards) and *Broke* (total exceeds 21). Aces have value 1 or 11 at the choice of the player. Using 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 [7] 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. The circuit should initially take an Ace as 11. This should be re-valued as 1 the first time the result would be *Broke*. But the given design continues to re-value the Ace card. Carefully simulating the circuit discovered a problem in the benchmark with one of the flag registers that indicates an Ace should be 11. By slightly modifying the circuit to remove the cause of a short duration pulse, the circuit was enabled to pass the test suite successfully.

## 4.3 Asynchronous FIFO

As a typical asynchronous circuit, an asynchronous FIFO buffer was specified and analysed. The FIFO has two inputs *InT, InF* and two outputs *OutT, OutF*. Its inputs and outputs use dual-rail encoding in which one bit needs two signal lines. The *Req* input comes from the environment of a stage, indicating that the environment has valid data to transfer. The *Ack* output goes to the environment, indicating that the stage is empty and ready for new data. The implementation uses two C-Elements (transition synchronisers used in asynchronous circuits). To ensure the FIFO works correctly, the environment has to be coordinated. It is convenient to think about the environment in two parts: *EnvF* (front-end) produces data, while *EnvB* (back-end) consumes it.

It was verified that the specification satisfies the following property: if there is an input of *1*, then the output will eventually become *1* (and similarly for input/output of *0*). It was verified that *Spec* $\approx$ *Impl* $||$ *(EnvB* $|[\cdots]|$ *EnvF)*, where $\approx$ denotes observational equivalence.

When speed independence needs to be verified, each building block (including the environment) should be specified in the input quasi-receptive style (*_QR*). It was 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)*.

It was also 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 |
| $\delta$ | InF !0 | OutF !0 | InT !1 | OutT !1 | InT !0 | InT !1 |
| OutT !0 | OutT !1 | $\delta$ | InT !0 | OutT !0 | $\delta$ | Pass |

## 4.4 Selector

A selector (an asynchronous design component) allows non-deterministic choice of output. After a change on input *Ip*, output *Op1* or *Op2* may change depending on the implementation. The *TestGen* tool produces a single test case of length 11 for the selector.

This example shows how test 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) | $\delta$ | Ip ! 1 |
| Op2 !1 ($\star$S1) | $\delta$ | 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, 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.

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.

## 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. 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.
4. 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.
5. Ji He and K. J. Turner. DILL (Digital Logic in LOTOS) project web page. `http://www.cs.stir.ac.uk/~kjt/research/dill.html`, Nov. 2000.
6. 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.
7. D. Winkel and F. Prosser. *The Art of Digital Design*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1980.