Specification and Verification of Synchronous Hardware using LOTOS. In Jianping Wu, Samuel T. Chanson, Quiang Gao, editors, Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX), pages 295-312, Kluwer Academic Publishers, London, UK, October 1999.

Specification and Verification of Synchronous Hardware using LOTOS

Ji He and Kenneth J. Turner
Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, Scotland

Keywords: Digital Logic, Hardware Description, LOTOS, Verification

Abstract: This paper investigates specification and verification of synchronous circuits using DILL (Digital Logic in LOTOS). After an overview of the DILL approach, the paper focuses on the characteristics of synchronous circuits. A more constrained model is presented for specifying digital components and verifying them. Two standard benchmark circuits are specified using this new model, and analysed by the CADP toolset (Cæsar/Aldébaran Development Package).

1. INTRODUCTION

1.1 Background

DILL (Digital Logic in LOTOS [14,16,17,25]) is an approach for specifying digital circuits using LOTOS (Language Of Temporal Ordering Specification [12]). DILL offers higher-level abstractions for describing hardware using a macro library for typical components and designs. DILL is used to formally specify digital hardware, using LOTOS at various abstraction levels. DILL addresses functional and timing aspects, supported by a library of common components and circuit designs, and using standard LOTOS tools.

The new work reported here allows synchronous circuits to be specified and verified. Two hardware verification benchmarks are used as examples. The paper extends the applicability of LOTOS in hardware design, and so is of interest to the LOTOS community. Of necessity some background in

LOTOS and hardware is required. The paper demonstrates the possibility of hardware verification using LOTOS, although some limitations will be discussed.

LOTOS supports rigorous specification and analysis, unlike semi-formal HDLs (Hardware Description Languages) such as VHDL (VHSIC Hardware Description Language [10]). LOTOS is neutral with respect to whether a specification is to be realised in hardware or software, allowing hardware-software co-design [22]. LOTOS inherits a well-developed verification theory from the field of process algebra, and has a theory for testing and test derivation. There is good support from general-purpose LOTOS toolsets such as CADP (Cæsar/Aldébaran Development Package [7]), LITE (LotoSphere Integrated Tool Environment) and LOLA/TOPO (LOTOS Laboratory). Most of these tools have been used in analysing DILL specifications. DILL is actually realised through translation into LOTOS.

This paper elaborates a DILL approach for modelling and verifying synchronous circuits. Synchronous design is chosen here 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 [16] for hardware timing analysis, using ET-LOTOS [19] as a basis.

Section 2 discusses how DILL models hardware, particularly synchronous circuits. Two case studies then demonstrate that DILL can successfully specify and verify standard benchmark designs. The Single Pulser in section 3 ensures that a switch causes well-defined pulses. The Bus Arbiter in section 4 grants bus access to only one client at a time among several.

1.2 Hardware Description and Verification

Hardware description has been studied extensively. Languages such as VHDL, Verilog [11] and ELLA [2] are commonly used in industry. These languages are semi-formal because their semantics is based on simulation models. Other HDLs do have formal semantics, e.g. CIRCAL (Circuit Calculus [21]), HOL [9] and Ruby [18]. 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. However, CIRCAL appears to be less effective at higher levels. For example, describing the behaviour of a synchronous circuit in CIRCAL requires the corresponding Mealy or Moore machine to be defined and then translated into CIRCAL.

Much of the early work on hardware verification used theorem-proving. Although quite general, this requires a significant amount of human guidance during verification. More recently model-checking, language containment and reachability analysis have attracted attention. Approaches using FSM (Finite State Machine) models can be automated, but they do not yet scale up to realistic hardware designs. The trend is 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.

Various researchers have studied the use of LOTOS for hardware description. The initial work at Stirling [25] overlapped independent work in Ottawa [6]. The European project FORMAT [5] studied the translation of LOTOS to VHDL. Other hardware applications of LOTOS have included bus protocols [3,23] and hardware synthesis [27].

The new DILL model for synchronous circuits has been evaluated on two standard benchmark circuits [24] that are intended for comparing different approaches to hardware verification. The machine used by the authors for verification was a SUN (300 MHz CPU, 128 MB memory).

1.3 Verification with CADP

The authors used CADP to verify DILL hardware. CADP accepts full standard LOTOS, using Cæsar.ADT for the data part of LOTOS and Cæsar for the behavioural part. The result is an LTS that can be used for verification. Aldébaran performs verification using the LTS or a network of LTSs (i.e. a finite state machine connecting several LTSs by LOTOS parallel and hiding operators). XTL (Executable Temporal Language) is a functional-like programming language that allows compact implementation of temporal logic operators. Several temporal logics such as ACTL (Action-based Computational Temporal Logic [4]) have been embedded in XTL. To partially solve the problem of state space explosion, CADP uses advanced verification techniques such as compositional generation, on-the-fly comparison, and a BDD (Binary Decision Diagram) representation of LTSs. These techniques permit verification of relatively large specifications.

CADP supports verification through bisimulation and temporal logic property checking. For verifying DILL (LOTOS) specifications, ACTL is an obvious candidate because the semantics of LOTOS is also based on actions. ACTL is also more understandable than the μ-calculus. The modal operators of HML (Hennessy-Milner Logic) are also employed in verification for

convenience. The subset of temporal operators used later in the paper is as follows. *A, B* and *C* are action sets, while *F* and *G* are formula sets.

ACTL_NOT_TO_UNLESS (A, B, C): this can be read as 'not *A* to *B* unless *C*'. After an action satisfying *A* in the current state, all paths leading to an action satisfying *B* must also satisfy *C*.

AG (F): all reachable states must satisfy *F*.

AU_A_B (F, A, B, G): this is the *until* operator $\mathcal{U}$. A restricted form is used in this paper: **AU_A_B** (**true**, *A*, *B*, **true**). This means that for the current state, each of its paths should have the following property: the actions along the path satisfy *A until* there is an action that satisfies *B*.

BOX (A, F): for the current state, all outgoing actions (if any) that satisfy *A* must result in states satisfying *F*.

EVAL_A (A): yields a state set corresponding to action *A*.

EX_A (A, F): from the current state, there exists an *A* that can lead to a state satisfying *F*.

WDIA (A, F): from the current state there exists a path with possible preceding internal actions and *A*, leading to a state satisfying *F*.

2. MODELLING APPROACH

2.1 General Approach

The basic philosophy of DILL is that it should be easy for the hardware engineer to translate a circuit schematic into a LOTOS specification, and then to analyse and verify the properties of this specification. There is thus a need for a component library. The library is available online for research purposes [15] and is summarised in Table 1.

It is possible to describe logic designs at different levels of abstraction, and to compare a higher-level design with a more detailed one. DILL does not give refinement guidelines, since these will be motivated by normal hardware design procedures. Components in the original DILL library were specified by progressively combining simpler components. This approach is termed structural since it reflects how a component is constructed. As the philosophy of DILL is to enable circuits, including library components, to be specified at different abstraction levels, higher level specification is also needed. This is termed the behavioural style. It specifies only what the component should do, not how it is constructed. Adding a new component to the DILL library does, of course, need reasonable knowledge of LOTOS. However the existing library provides simple patterns to follow as examples.

Having abstract (behavioural) as well as design (structural) specifications in the library is helpful in both bottom-up and top-down design.

Table 1. DILL Library

Component	Variants
Adder	2/4 inputs, behavioural/structural, half/full/parallel/ripple
And, ...	2/3/4//8 inputs, 0/1-active tri-state enable
Clock	-
Comparator	1/4/8/*n* inputs, behavioural/structural
Counter	behavioural/structural
Decoder	2/3 inputs, behavioural/structural, 0/1-active outputs, BCD/Decimal/Excess-3/Gray
Demultiplexer	1/2 inputs, behavioural/structural
Delay	dynamic/general/hold/inertial/pure/setup/width/edge
Divider	2/4/8 inputs, behavioural/structural, positive/negative edge trigger
Encoder	4/8 inputs, behavioural/structural, 0/1-active outputs
Flip Flop	D/JK/MS/RS/T, behavioural/structural, positive/negative edge trigger, preset, preclear, lockout
Inverter	1/4/8 inputs, 0/1-active tri-state enable
Latch	D/RS, 1/4/8 bits, behavioural/structural, preset, preclear, clocked
Memory	behavioural/structural
Multiplexer	2/4 inputs, 1/8/*n*-bit, behavioural/structural
One, ...	source of logic 1/0, sink
Parity	8 inputs, behavioural/structural
Register	4/8/*n* bits, behavioural/structural, positive/negative edge trigger, load enable/preclear, tri-state output, bucket brigade/pass-on/shift
Repeater	/4/8 inputs, 0/1-active tri-state enable

Since component specifications are translated into LOTOS, the designer must be familiar with how to combine LOTOS behaviour expressions. Fortunately the relationship between a circuit design and its DILL representation is straightforward, and does not require detailed LOTOS knowledge. The principal method of connecting components is to compose their behaviours in parallel. The synchronisation rules of LOTOS allow components to be connected in a natural way.

LOTOS, like most specification languages, deals only with discrete events. It is therefore signal changes that are modelled in asynchronous (unclocked) design. However 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, and so correspond to one LOTOS event per clock cycle in the synchronous case.

Wires or tracks between components are not normally represented explicitly in DILL. A component's ports (e.g. its pins) are represented by LOTOS gates. (The term 'gate' will be qualified as it has different meanings in hardware and LOTOS.) and 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. Multi-bit signals or multi-wire connections (e.g. buses) and multi-component assemblies (e.g. memory arrays) are supported by DILL.

2.2 Synchronous Circuit Model

A piece of combinational logic merely combines its inputs to produce outputs; it is referred to as a stage in the following. Sequential logic incorporates feedback, so the state of an output depends on previous inputs. Synchronous circuits, as one form of sequential design, are distinguished from asynchronous circuits through control by a global clock.

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 determine the state of the whole circuit.

For a synchronous circuit, the designer 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. As discussed in [16], Timed DILL *can* specify such constraints. However, sections 2.4 and 3 will show that properly modelling the storage components and environment ensures a DILL specification always meets the clock condition.

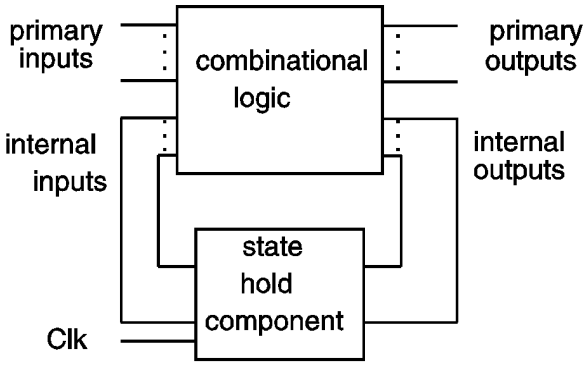

Figure 1. Synchronous Circuit Model

In synchronous design, the primary inputs are usually synchronised with the clock signal. This eases design and analysis of synchronous circuits. DILL incorporates this practice into its synchronous circuit model, assuming that the primary inputs have already been synchronised with the clock signal.

Besides the above, the DILL synchronous model has two more 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 way components are modelled, for otherwise a DILL specification might deadlock where a real circuit could still work. This is discussed further in section 2.4 .

2.3 Synchronous Model for Basic Logic Gates

The fundamental DILL model for basic logic gates allows an input or output port to offer an event corresponding to a signal change at any time. This model is a very generic representation of logic gates used in real world, but this may lead to non-determinism due to the lack of quantified timing [17]. The gate model therefore has to be constrained according to the environment in which the gates operate. Logic gates are presumed to be part of a synchronous design. 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 transient values are ignored because they do not affect circuit behaviour. The synchronous model allows basic logic gates (and thus all other components within combinational logic) to wait until all inputs occur before outputting the corresponding value.

The following example models a two-input *nand* gate. Note that inputs are interleaved, i.e. they can occur in any order. It might appear that the order of input events could be fixed since it does not influence the functionality of a component. This would result in a smaller state space when circuits are verified. Unfortunately this might cause deadlock when components are connected. Suppose that components *A* and *B* each have two inputs. Imagine that inputs are required in the order *IpA1* before *IpA2*, and *IpB1* before *IpB2*. This would lead to deadlock if the components shared inputs, with *IpA1* connected to *IpB2* and *IpA2* connected to *IpB1*. For this reason, DILL insists on fully interleaved inputs.

```
process Nand2 [Ip1, Ip2, Op] : noexit :=
  (Ip1 ?dtIp1 : Bit; exit (dtIp1, any Bit)                    (* allow one input *)
  |||
    Ip2 ?dtIp2 : Bit; exit (any Bit, dtIp2))                  (* allow other input *)
>> accept dtIp1, dtIp2 : Bit in                               (* accept both inputs *)
  (Op !(dtIP1 nand dtIp2);                                    (* output nand of inputs *)
    Nand2 [Ip1, Ip2, Op])                                     (* repeat behaviour *)
endproc (* Nand2 *)
```

2.4 Synchronous Model for State Hold Components

The gate model just discussed is not suitable for circuits with cyclic connections since these result in input-output interdependency and thus in specification deadlock. Cyclic connections are common in latches and flip-flops, so state hold components are modelled in the behavioural style. At a higher level of specification and design, problems due to cyclic connections do not arise. For synchronous circuits, two modifications are made to the fundamental DILL model. LOTOS events are considered to model signal levels rather than changes, and a constraint is added to reflect the assumption of a slow enough clock. A DFF (Delay Flip-Flop) is a simple memory element with data input *D*, clock input *Clk* and output *Q*. Its specification is as follows:

```
process DFF [D, Clk, Q] (dtD, dtClk : Bit) : noexit :=
  D ?newdtD : Bit; DFF [D, Clk, Q] (newdtD, dtClk);          (* input new data *)
  Clk ?newdtClk : Bit;                                       (* input clock pulse *)
 ([(dtClk eq 1) and (newdtClk eq 0)] →                       (* ignore negative pulse *)
    DFF [D, Clk, Q] (dtD, newdtClk)                          (* continue behaviour *)
  []
    [(dtClk eq 0) and (newdtClk eq1)] →                      (* react to positive pulse *)
      Q !dtD;                                                (* output stored data *)
      DFF [D, Clk, Q] (dtD, newdtClk)                        (* continue behaviour *)
    )
endproc (* DFF *)
```

Suppose a combinational logic circuit feeds into this flip-flop as the state hold component. If the clock signal is not constrained, it is possible that the clock moves to the next cycle before the combinational logic has settled down. The model of a synchronous circuit must exclude this possibility. After a positive-going transition of the clock signal, if the *D* input of the flip-flop has not occurred yet then the next positive-going transition of clock signal must not occur. This is ensured by the following constraint on the D flip-flop specification. The process *Cons_DFF* deals with the initial state of the flip-flop The next positive-going clock transition is handled by process *Cons_DFF_Aux*. The full specification of a D flip-flop combines *DFF* and *Cons_DFF* with the LOTOS parallel operator.

```
process Cons_DFF [D, Clk] (dtClk : Bit) : noexit :=
  D ?newdtD : Bit;                                           (* input new data *)
  Cons_DFF [D, Clk] (dtClk)                                  (* continue behaviour *)
[]
  Clk ?newdtClk : Bit;                                       (* input clock pulse *)
  ([(newdtClk eq 1) and (dtClk eq 0)] →                      (* react to positive pulse *)
    Cons_DFF_Aux [D, Clk] (newdtClk)                         (* after one clock pulse *)
[]
  [(newdtClk eq 0) and (dtClk eq 1)] →                       (* ignore other pulses *)
    Cons_DFF [D, Clk] (newdtClk))                            (* continue behaviour *)
where
```

```
  process Cons_DFF_Aux [D, Clk] (dtClk : Bit) : noexit :=
    D ?newdtD : Bit; Clk !0; Clk !1;                (* input before negative pulse  *)
    Cons_DFF_Aux [D, Clk] (1)                              (* continue behaviour *)
  []
    Clk !0; D ?newdtD : Bit; Clk !1;                 (* input after negative pulse *)
    Cons_DFF_Aux [D, Clk] (1)                              (* continue behaviour *)
  endproc (* Cons_DFF_Aux *)
endproc (* Cons_DFF *)
```

3. CASE STUDY: A SINGLE PULSER

The informal description of the Single Pulser appears in the standard benchmark document [24]. A Single Pulser is a clocked-sequential device with a one-bit input *I* and a one-bit output *O*. It deals with a debounced switch that is on (true) in the down position and off (false) in the up position. When the Single Pulser senses the switch being turned on, it must assert an output signal lasting one clock cycle. The circuit should not allow additional outputs until after the switch has been turned off. The benchmark also informally defines some properties that the Single Pulser must respect.

3.1 Specification

Figure 2 shows a design for the Single Pulser given in the benchmark. *P_In* is the input from the switch, and *P_Out* is the output from the circuit. It is very straightforward to represent the Single Pulser design in DILL. Because the clock is implicit in a synchronous circuit design, circuit properties may not actually refer to it. Experience shows that hiding the clock signal can make the temporal logic formulae clearer. The Single Pulser specification is as follows (omitting process gate names for brevity):

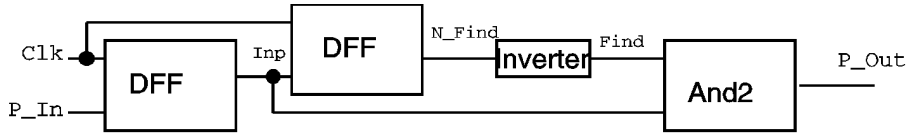

Figure 2. Single Pulser Design

```
hide Inp, N_Find, Find, Clk in                          (* hide internal gates *)
  ((Cons_DFF  |[ N_Find, Inp ]|  (Inverter  |[ Find ]|  And2))   (* flip-flop, inverter, and *)
  |[ Clk, Inp ]|                                        (* synchronised with ... *)
   Cons_DFF)                                                (* flip-flop *)
|[ P_In, Clk, P_Out ]|                                  (* synchronised with ... *)
   Env                                                   (* the environment *)
```

The *Env* process serves as the environment constraint on the Single Pulser. It permits *P_In* to come before each positive-going clock transition, and allows the next clock cycle only after *P_Out* has occurred. Without this constraint, the properties discussed later are invalid. The constraint between *P_In* and *Clk* ensures that *P_In* is synchronised with *Clk*. The constraint between inputs and output respects the slow-clock requirement: *P_Out* must happen before the next positive-going clock transition. These assumptions are not automatically guaranteed by the circuit design, but they are required by the DILL synchronous circuit model. In outline, *Env* is specified as:

```
(P_In ? dtPIn : Bit;                                    (* pulse in *)
 Clk ! 1;                                               (* positive-going clock *)
 (Clk ! 0; exit  |||  P_Out ? dtPOUt : Bit; exit))      (* negative-going clock, pulse out *)
>>                                                      (* and then ... *)
 Env                                                    (* same environment behaviour *)
```

3.2 Verification

The formulation of properties in CADP was briefly explained in section 1.3 For brevity the properties are given only informally here; see the details in [17]. Verification of the Single Pulser was undertaken using only XTL model checking, although it is not difficult to give a higher level specification in DILL/LOTOS and then check for equivalence between the two levels. Because LOTOS events are modelled as signal levels instead of signal transitions, representing a rising edge needs two clock cycles. In the first cycle the signal should be at level 0, in the second cycle it should be at level 1. Each signal happens once and only once in a clock cycle, so the second appearance of the same signal indicates the second clock cycle.

Property 1: If *P_In* has a rising, eventually *P_Out* becomes true.

Property 2: Whenever *P_Out* is 1, it becomes 0 in the next state and remains 0 at least until the next rising edge on *P_In*.

Property 3: Whenever there is a rising edge, and assuming that the output pulse does not happen immediately, there are no more rising edges until that pulse happens. In other words, there cannot be two rising edges on *P_In* without a rising edge on *P_Out* between them.

The size of the LTS produced by Cæsar.ADT and Cæsar from the DILL specification has 295 states and 538 transitions. Aldébaran minimises the LTS to a smaller one having 97 states and 174 transitions modulo strong bisimulation. Because the resultant LTS is small, all the generation and verification steps take negligible time. Aldébaran uses the LTS to show that the DILL design is deadlock free. The XTL tool is also able to demonstrate that all the supposed properties of the circuit are valid.

4. CASE STUDY: A BUS ARBITER

In this section, the DILL approach is evaluated using another benchmark circuit. For brevity, the specifications are not given here but can be found in [17]. The purpose of the Bus Arbiter is to grant 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, each from a client. The outputs are a set of acknowledge signals, indicating which client is granted access during a clock cycle. The documentation also defines some properties that the Bus Arbiter must respect. Thse are given informally and also in CTL (Computational Temporal Logic). Besides listing the properties to be fulfilled, the benchmark documentation also gives an arbitration algorithm in plain English. Finally the gate level implementation of the Bus Arbiter is provided as a circuit diagram.

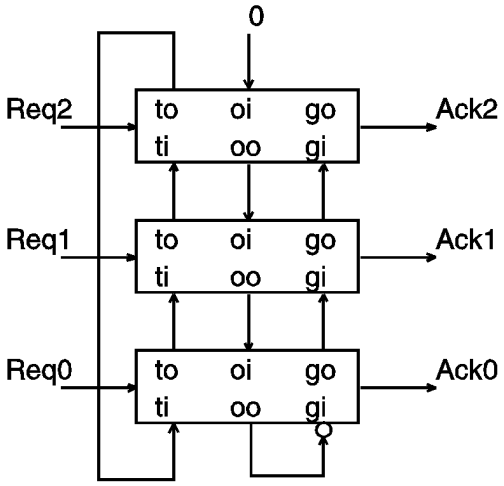

Figure 3. Bus Arbiter With Three Cells

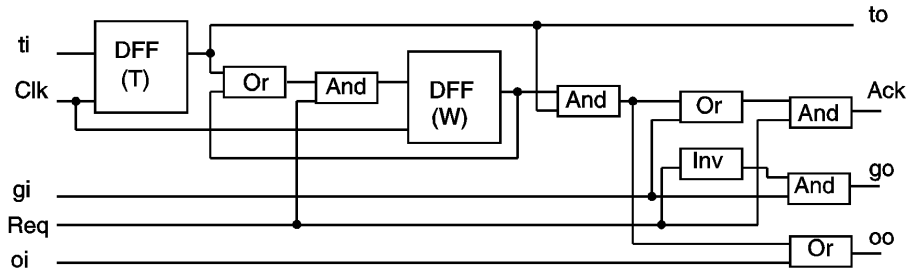

Figure 4. Design of An Arbiter Cell

4.1 Higher-Level Specification in LOTOS

LOTOS supports specification at various levels of abstraction. Although the benchmark circuits have been studied by many researchers, as far as the authors knowledge 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. There are two clear benefits of this formalisation. Firstly, better understanding of the algorithm can be gained from rigorous specification. Secondly, correctness of the algorithm itself can be ensured before the circuit is built and verified. Flaws in the algorithm will be more time-consuming to fix if they are found only after implementation.

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.

Translating the algorithm to LOTOS is quite straightforward, mainly using LOTOS value expressions. For example each cell has two associated variables: *token* indicates if the token is in the cell, and *waiting* indicates if the client's request has persisted for a completed token cycle. Circulating the token, (re)setting the waiting variable and so on correspond to LOTOS value expressions. For an arbiter with three cells, the LOTOS specification has 79 lines (including comments) for the behavioural specification.

4.2 Lower-Level Specification in DILL

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. Figure 3 shows an arbiter with three cells. Figure 4 shows the design of each cell. The first cell is slightly different because it is assumed that the token is initially in the first cell.

The principle of the circuit will not explained in detail here. Briefly, the *ti* (token in) and *to* (token out) signals are for circulation of the token. The *to* output of the last cell connects to the *ti* input of the first cell to form a ring. The *gi* (grant in) and *go* (grant out) signals are related to priority. The grant of cell i is passed to cell $i+1$, meaning no client of index $\leq i$ is requesting. Hence a cell may assert its acknowledge output if its grant input is asserted. The *oi* (override in) and *oo* (override out) signals are used to override the

priority. When the token is in a persistent requesting cell, its corresponding client will get access to the bus. The *oo* signal of the cell is set to 1. This signal propagates down to the first cell and resets its grant signal through an inverter. As a consequence the *gi* signal of every cell is reset, in other words the priority has no effect during this clock cycle. Within each cell, register *T* stores 1 when the token is present; register *W* (waiting) is set to 1 when there is a persistent request. Initially the token is assumed to be in the first cell.

The components of each cell are in the DILL library, so specification of a cell is very easy. The specification of an arbiter with three cells is obtained by connecting three such processes. As for the Single Pulser, there is also an environment constraint in the structural specification of the arbiter to meet the conditions of the synchronous circuit model discussed in section 2.2.

Since the properties that the arbiter must fulfill are given in the benchmark documentation, it is obvious that the verification should consist of model checking these properties. Equivalence checking is also performed since two levels of specifications are identified.

4.3 Verification

Section 1.3 explained how to formulate properties in CADP. They are translated into action-based temporal logic (ACTL and HML). The following properties refer to client 0; the formulae for other clients have a similar form.

Property 1: No two acknowledge outputs are asserted in the same clock cycle (safety).

```
AG (                                              (* for all states ... *)
 not (                                            (* it is not the case that ... *)
  EX_A (                                          (* there exists action *)
   EVAL_A (Ack0 !1)                               (* Ack0 !1 leading to ... *)
    (WDIA (EVAL_A (Ack1 !1), true) or             (* action Ack1 !1 or *)
     WDIA (EVAL_A (Ack2 !1), true)))))            (* action Ack2 !1 *)
```

Property 2: Every persistent request is eventually acknowledged (liveness).

```
AG (                                              (* for all states ... *)
 BOX (                                            (* after all its outgoing action *)
  EVAL_A (Req0 !1),                               (* which is Req0 !1 ... *)
   AU_A_B (true, true,                            (* until ... *)
    (EVAL_A (Ack0 !1) or                          (* eventually Ack0 !1 ... *)
     EVAL_A (Req0 !0)), true)))                   (* unless Req0 !0 *)
```

Property 3: Acknowledge is not asserted without request (safety).

```
AG (                                              (* for all states *)
 ACTL_NOT_TO_UNLESS (                             (* not Req0 !0, Ack0 !1 unless Req0!1 *)
  EVAL_A (Req0 !0),                               (* after Req0 !0 *)
   EVAL_A (Ack0 !1),                              (* Ack0 !1 is impossible ... *)
    EVAL (Req0 !1)))                              (* unless after Req0 !1 *)
```

To verify the higher-level specification against the temporal logic formulae, the LTS of the specification was produced first. Cæsar generates an LTS with 3649 states and 7918 transitions. Aldébaran reduces this to 379 states and 828 transitions with respect to strong bisimulation. Both generation and reduction take seconds. The temporal logic formulae are then checked against the minimised LTS. Each is verified as true within 1 minute.

The real challenge comes when the lower-level DILL specification is verified. The state space is so large that direct generation of the LTS from the LOTOS specification is impractical. As mentioned before, there are several advanced techniques implemented in CADP to tackle the problem of state space explosion. Nevertheless, using on-the-fly verification of the arbiter also fails after considerable run-time. CADP does not currently support the direct generation of BDDs from a LOTOS specification.

Compositional generation was tried out to verify the arbiter. Basically the idea is that of 'divide and conquer'. A LOTOS specification is divided into several smaller specifications to make sure that it is possible for Cæsar to generate an LTS for each of them. Then Aldébaran is used to reduce these LTSs with respect to a suitable equivalence relation. The minimised LTSs are then combined using the LOTOS parallel operator (and also the hide operator if necessary) to form a network of communicating LTSs (the CADP term). At this stage, an LTS might be produced from the network, or on-the-fly verification might be performed against the network. In order to get valid verification results, special attention must be given to the equivalence relation that is used. The relation must be a congruence at least with respect to the compositional operators, here the LOTOS parallel and hide operators. The relation must also preserve the properties to be verified. This ensures that the resulting network of communicating LTSs will respect the same properties as the original LOTOS specification.

Among the benchmark properties, the first and the third concern safety while the second concerns liveness. Safety equivalence [1] preserves safety properties, while branching bisimulation equivalence [26] preserves liveness properties when there are no livelocks in specifications. Both of these equivalences are congruences with respect to the parallel and hide operators. These two equivalences are thus appropriate to compositional generation.

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

Before checking equivalence, a suitable equivalence must be chosen. For most systems, observational equivalence is an obvious choice. Informally it means that two systems have exactly same behaviour in terms of the observable actions. For hardware systems, testing equivalence (two specifications pass or fail exactly the same external tests) is also used as a criterion in some approaches such as CIRCAL. The algorithm for testing equivalence is not implemented in CADP, so the stronger notion of observational equivalence was used when checking the Bus Arbiter.

As before, compositional generation was used to generate the LTS for the design. This time each cell was reduced with respect to observational equivalence, since this is a congruence for the parallel and hide operators. Generating the LTS took about eight minutes. This LTS was expected to be observationally equivalent to that for the higher-level specification. However Aldébaran discovered that they are not! Table 2 is one of the sequences given as a counter-example. (The Aldébaran output has been rendered more readable here.) This sequence indicates that in the first three clock cycles only client 0 requests the bus; both the high-level specification and the low-level design 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 lower-level specification offers 0 for *Ack1*, whereas the higher-level specification offers 1 for *Ack1*.

Table 2. A Counter-Example generated by Aldébaran

	Cycle 1	**Cycle 2**	**Cycle 3**	**Cycle 4**
Req0	1	1	1	0
Req1	0	0	0	0
Req2	0	0	0	0
Ack0	1	1	1	
Ack1	0	0	0	**0** or **1**
Ack2	0	0	0	

After step-by-step simulation of the counter-example, it was soon discovered that the circuit may not properly reset the *oo* (override out) signal to *0*. Suppose a cell has been requesting access, so its *W* register is set to *1*. If the cell cancels the request in the very clock cycle that the token happens to arrive. In this situation, because the client has already cancelled its request it should be possible for another client to get the bus. However, the design sets the *oo* signal to override the priority as if this client were still requesting. This prevents any other client from accessing the bus in this clock cycle.

Fixing the problem was much easier than finding it. The correction was to connect the *Req* signal to the *And* gate that follows the *W* register. (See [17] for the revised circuit diagram.) The output of the *And* gate guarantees that the *oo* signal is always correctly set or reset according to the request

signal in the current clock cycle. This modified design was verified to be observationally equivalent to the higher-level algorithmic specification.

As mentioned in section 2.2, in DILL the inputs are assumed to be synchronised with the clock signal. Suppose that the *Req* signal in figure 4 is always ready before the active clock transition, i.e. is not synchronised with the clock. In this case the problem discussed above might not happen. As the benchmark documentation does not state if inputs are synchronised with the clock or not, it is believed that the modified design is more robust.

5. CONCLUSION

With the new approach to specifying synchronous, it is possible to verify standard hardware benchmarks – here, the Single Pulser and the Bus Arbiter. In comparison with other techniques applied to the same case studies, e.g. COSPAN [8] and CIRCAL [20], DILL is much more convenient for giving a higher-level specification. This is not so surprising since LOTOS is an expressive language. CIRCAL, by way of contrast, gives an abstract view of a synchronous circuit by directly specifying its corresponding finite state machine, which 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. The former needs human assistance to complete a proof. The latter needs specialised expertise since temporal logic specifications are not easy to write. In contrast, 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. Another benefit of equivalence checking is seen in the Bus Arbiter case study. As a classical verification benchmark, the Bus Arbiter has been investigated using many approaches. But as far as the authors know, the defect reported in section 4.3 is a new discovery.

However, the size of the circuit that can be effectively verified is small compared to that handled by other mature hardware verification tools. COSPAN can verify an arbiter with four cells with the consumption of about 1 MB memory, due to a symbolic representation using BDDs and efficient reduction techniques [8]. CIRCAL is reported to generate the state space of an arbiter with up to 40 cells using reasonable computing resources, although the actual memory used was not reported [20]. Again this is due to the BDD representation of the CIRCAL specification. Note that CIRCAL was not in fact used to verify the arbiter formally. [20] just gives a test pattern to show that even if all clients request the bus, only one can gain access to the bus in each

clock cycle. CIRCAL does not have the functionality of temporal logic model checking. Because of its limited power in specifying higher-level behaviour, equivalence checking was not used in the CIRCAL case study. CADP on the other hand consumes more than 100 MB of memory to produce the state space of a three-cell arbiter. Although the resulting state space is relatively small, the intermediate stages of generation need considerable memory.

There are two main reasons for this performance limitation. One comes from the modelling language LOTOS and the other comes from CADP. Firstly, 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. Secondly, CADP is a tool under development and currently some of its features are mainly based on explicit state exploration. Because CADP cannot produce the minimised state space in the first place, large amounts of memory have to be consumed before a smaller LTS can be produced by minimisation. On-the-fly algorithms are of some help, but they apply only in particular situations. For example, on-the-fly observational equivalence checking is not supported by CADP. Also CADP does not offer a BDD representation of LOTOS specifications, although BDDs are used to represent intermediate data types in some algorithms. Fortunately CADP is currently being actively improved by the CADP developers.

REFERENCES

[1] A. Bouajjani, J. C. Fernandez, *et al*. Safety for branching time semantics. In *Automata, Languages and Programming*, LNCS 510, pages 76–92. Springer-Verlag, Berlin, 1991.

[2] R. Boulton, M. J. C. Gordon *et al*. The HOL verification of ELLA designs. TR 199, University of Cambridge Computer Laboratory, Aug. 1990.

[3] G. Chehaibar, H. Garavel, *et al*. Specification and verification of the PowerScale bus arbitration protocol: An industrial experiment with LOTOS. TR 2958, INRIA, Le Chesnay, Aug. 1996.

[4] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *Semantics for Systems of Concurrent Processes*, LNCS 469, pages 407–419. Springer-Verlag, Berlin, 1990.

[5] C. Delgado Kloos, T. de Miguel *et al*. VHDL generation from a timed extension of the formal description technique LOTOS with the FORMAT project. *Microprocessing and Microprogramming*, 38:589–596, 1993.

[6] M. Faci and L. M. S. Logrippo. Specifying hardware in LOTOS. In *Proc. Computer Hardware Description Languages and Their Applications XI*, pages 305–312. North-Holland, Amsterdam, Apr. 1993.

[7] J.-C. Fernández, H. Garavel, *et al*. CADP (Cæsar/Aldébaran development package): A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors,

Proc. Computer-Aided Verification VIII, LNCS 1102, pages 437–440. Springer-Verlag, Berlin, Aug. 1996.

[8] K. Fisler and R. P. Kurshan. Verifying VHDL designs with COSPAN. In *Formal Hardware Verification Methods and Systems in Comparison*, LNCS 1287, pages 206–247. Springer-Verlag, Berlin, 1997.

[9] C. A. R. Hoare and M. J. C. Gordon, editors. *Mechanized Reasoning and Hardware Design*. Prentice Hall, Englewood Cliffs, 1992.

[10] IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, 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, 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, 1989.

[13] ISO/IEC. *Information Processing Systems – Open Systems Interconnection – Enhancements to LOTOS*. International Organization for Standardization, Geneva, Apr. 1998.

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

[15] Ji He and K. J. Turner. DILL (Digital Logic in LOTOS) translator. *http://www.cs.stir.ac.uk/~kjt/software/dill.html*, Jan. 1998.

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

[17] Ji He and K. J. Turner. Modelling and verifying synchronous circuits in DILL. TR CSM-152, Computing Science and Mathematics, University of Stirling, Feb. 1999.

[18] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods For VLSI* Design, pages 13–70. Elsevier Science Publishers, Amsterdam, 1990.

[19] 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.

[20] G. A. McCaskill and G. J. Milne. Sequential circuit analysis with a BDD based process algebra system. TR HDV-25-93, Computer Science, University of Strathclyde, Jan. 1993.

[21] G. J. Milne. *The Formal Specification and Verification of Digital Systems*. McGraw-Hill, New York, 1994.

[22] L. Sánchez Fernández, M. L. López *et al*. Co-design at work: The Ethernet bridge case study. *Current Issues in Electronic Modelling*, 8, Apr. 1996.

[23] M. Sighireanu and R. Mateescu. Validation of the link layer protocol of the IEEE-1394 serial bus ('Firewire'): An experiment with E-LOTOS. TR 3172, Institut National de Recherche en Informatique et Automatique, Le Chesnay, May 1997.

[24] J. Staunstrup and T. Kropf. IFIP WG10.5 benchmark circuits. *http://goethe.ira.uka.de/hvg/benchmarks.html*, July 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, 1994.

[26] R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation. TR CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989.

[27] K. Yasumoto, A. Kitajima *et al*. Hardware synthesis from protocol specifications in LOTOS. In S. Budkowski, E. Najm, and A. Cavalli, editors, *Proc. Formal Description Techniques XI/Protocol Specification, Testing and Verification XVIII*. Chapman-Hall, London, 1998.