Ji He and Kenneth J. Turner. Specifying Hardware Timing with ET-LOTOS (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 161-166, Springer Verlag, Berlin, Germany, September 2001.

Specifying Hardware Timing Characteristics with ET-LOTOS

Ji He and Kenneth J. Turner

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

Abstract. It is explained how DILL (Digital Logic in LOTOS) can be used to specify and analyse hardware timing characteristics using ET-LOTOS (Enhanced Timed LOTOS), a timed extension of the ISO standard formal language LOTOS (Language of Temporal Ordering Specification). Hardware component functionality and timing characteristics are rigorously specified and then validated. As will be seen, subtle timing problems can be found by using this approach.

Keywords: DILL (Digital Logic in LOTOS), ET-LOTOS (Enhanced Timed LOTOS), formal method, LOTOS (Language of Temporal Ordering Specification), timing characteristic

1 Introduction

DILL (Digital Logic in LOTOS [2–8, 13]) is an approach for specifying digital circuits using LOTOS (Language Of Temporal Ordering Specification [1]). 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. The novelty of the paper is thus in the use of a language from the communications field in hardware design. 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.

The paper uses ET-LOTOS (Enhanced Timed LOTOS [9]). Since the readers of this paper are unlikely to be familiar with (ET-)LOTOS, a summary of the notation used in the paper is given in figure 1. Because ET-LOTOS tools are currently under development, the authors have also used TE-LOTOS (Time Extended LOTOS [12]) for validation.

The specification and analysis of communications systems with LOTOS is well understood. The work reported here on hardware timing is new. To explain the method clearly without the complication of a large problem, only a simple case study is given. However the method is generally applicable to all sizes and levels of hardware design.

LOTOS	Meaning
process P [gates] (parameters) : **noexit** := B1 **where** ... **endproc**	non-terminating process definition with subsidiary definition
P [gates] (parameters)	instance of process
let var : sort = value **in**	local variable definition
i	internal event
gate ? var : sort ! value	observable event at gate with input and output
event [condition]	condition on event
event {min, max}	range of times for event
event @var	record time of event in variable
event ; B	event then behaviour
delta (time)	delay
[guard] $\gg$ B	conditional behaviour
B1 $[]$ B2	choice of behaviours
B1 >> B2	behaviours in sequence
B1 ||| B2	parallel behaviours interleaved
B1 $\|[gates]\|$ B2	parallel behaviours synchronised on event gates
exit	successful termination
stop	deadlock

Fig. 1. Summary of (ET-)LOTOS Syntax

In particular, it is suitable for giving very high level timing characteristics (say, at the level of a system block diagram) and relating these to timing characteristics at intermediate levels of design.

It will be seen how Timed DILL can be used to specify and analyse digital designs that are time-sensitive. Tools supporting ET-LOTOS were under development during the work reported here. The authors therefore made use of TE-LOLA (Time Extended LOTOS Laboratory [11]), which supports TE-LOTOS (Time Extended LOTOS [12]).

It has been possible to use TE-LOLA to analyse the specifications generated by Timed DILL. Although ET-LOTOS and TE-LOTOS adopt different semantic models, the equivalence between them has been established [10]. It is therefore possible to translate the generated ET-LOTOS specifications into TE-LOTOS syntax. Because of their similarity, the translation is always possible although some subtle differences need attention. In order to avoid confusion, the following specifications will use ET-LOTOS though the actual analysis was made with TE-LOTOS.

2 Timed Specification Approach

2.1 ET-LOTOS

ET-LOTOS is a timed LOTOS that allows the modelling of time-sensitive behaviour. It supports both discrete and dense time domains. Three new operators relevant to time are introduced in ET-LOTOS: delay, life reducer and time measurement. The formal semantics of ET-LOTOS is given by a labelled transition system. There are two kinds

of transitions: discrete and timed. A discrete transition corresponds to the execution of an action and timed transitions correspond to the passage of time.

The delay operator *delta (time)* means that the subsequent behaviour will be delayed by *time*. In ET-LOTOS a time value is relative to the instant when the previous action occurs. The behaviour *e; delta (d); B* will delay for *d* after event *e* occurs and then behave like *B*. The time measurement operator *@t* is used to measure the time elapsed between the instant when the event has been offered and the instant when it occurs. The time value is stored in *t*.

The life reducer operator has different semantics when applied to internal events (**i**) and observable events (*e*). **i** $\{d\}$ means that **i** *must* occur non-deterministically within the next *d* time units. Necessity and non-determinism apply because internal actions are not controlled by the environment; in particular, the time of occurrence is decided by the system. In the case of observable behaviour *e* $\{d\}$*; B*, the event *may* happen within *d* time units. If so the behaviour evolves to *B*, otherwise the process deadlocks. The default life reducer for internal events is *0*, while for observable events it is the maximal value of the time domain.

ET-LOTOS adopts maximal progress for hidden actions, with important consequences for Timed DILL. Maximal progress means that if a hidden action can occur, it must happen at once (unless an alternative action occurs). In other words, hidden actions are urgent in ET-LOTOS. In DILL, each digital component is modelled as a process which is usually connected to others. Input or output ports are modelled by LOTOS event gates. Ports within a design are hidden, so their events become urgent under the assumption of maximal progress.

2.2 Modelling Timed Hardware

Before developing an abstract model to specify timed digital components, the required timing characteristics must be considered. These define timing relationships among inputs, outputs, and inputs/outputs. The timing relationship from input to output is normally called *delay*. It is the time interval between a signal change on an input and the resulting signal change on an output. A timing relationship among inputs is called a *timing constraint*, meaning that the digital circuit can work correctly only if the constraints are met. There is no need to specify the timing relationships among outputs directly, as they are determined by delays and timing constraints.

There are several possible approaches to specifying a timed digital component, classified as *integrated methods* or *combined methods*. In an integrated method, a digital component is specified in one process that deals with both functionality and timing. Although the integrated method may result in compact specifications, it is not a 'structural' method and is hard to apply. The approach is not compositional in the sense that functional and temporal characteristics of a component are not merely combined. It is also important to have untimed behaviour as a simple case of timed behaviour, i.e. to be able to isolate pure functionality. Attention has therefore been focused on developing combined methods. The idea is to separate the functionality and the timing characteristics into different processes, and then to combine them in an appropriate way.

The model adopted for Timed DILL was arrived at after considerable experimentation with different ideas. The selected approach is called the *parallel-serial* model. As

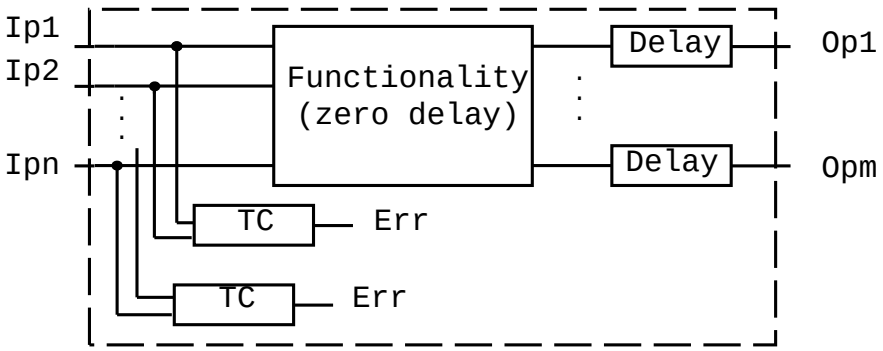

Fig. 2. The Specification Model for a Timed Component

shown in figure 2, the functionality is assumed to be specified with zero delay. Timing constraints (TC) are placed in parallel with the functional specification to check if input requirements are met. Delays are placed in series with the functionality to provide delay for each output. The sections that follow explain each aspect in more detail.

Note that the *Err* event gates in the figure are only for analysis of errors; they have no counterpart in a real physical component. If an *Err* action is offered, it indicates that a timing constraint has not been met. However, the behaviour of the component is not influenced by errors since the functionality part always assumes that inputs meet the constraints. The outputs of the component are thus always 'correct' in terms of the component's function.

This modelling decision was made after careful consideration. The idea is that instead of trying to specify behaviour under all kinds of input conditions, behaviour is specified only for correct inputs. Behaviour under error conditions is 'wrong' and thus not very meaningful. Violation of timing constraints means that there is a design error. Real hardware will do something under these conditions, but the result is not really interesting or relevant. It is more important to detect and correct design errors, not model the behaviour of components under erroneous conditions. The aim of analysis is to find out whether the design is correct, particularly in the presence of timing conditions. During a simulation, the occurrence of an *Err* offer is immediately obvious. For verification, the absence of *Err* offers can be checked before verifying other properties.

Finally, note that if the timing constraints are void and the delays are between zero and arbitrarily large then the timed model is equivalent to the untimed model. An untimed specification is thus just a special case.

3 Component Functionality

As shown in figure 2, the block dealing with component functionality is supposed to have zero delay. This block can be easily obtained from its untimed counterpart. To change an untimed specification to one with zero delay, a life reducer $\{0\}$ is appended to each output event offer. The following illustrates a two-input *and* gate:

```
process And2 [Ip1, Ip2, Op] (DataIp1, DataIp2, DataOp : Bit) : noexit :=
  Ip1 ? NewDataIp1 : Bit;                                  (* input 1 change *)
  And2 [Ip1, Ip2, Op] (NewDataIp1, DataIp2, DataOp)            (* continue *)
[]                                                                (* or ... *)
  Ip2 ? NewDataIp2 : Bit;                                  (* input 2 change *)
  And2 [Ip1, Ip2, Op] (DataIp1, NewDataIp2, DataOp)            (* continue *)
[]                                                                (* or ... *)
  (
    let NewDataOp : Bit = DataIp1 and DataIp2 in           (* set new output *)
      Op ! NewDataOp [NewDataOp ne DataOp] {0};          (* output at once *)
      And2 [Ip1, Ip2, Op] (DataIp1, DataIp2, NewDataOp)        (* continue *)
  )
endproc (* And2 *)
```

4 Delays

4.1 Basic Delay Types

Because the physical structures of digital components differ, their delays are of different types and values. There are two basic delay types: *pure delay* and *inertial delay*. Suppose the delay of a digital component is D. If a component has pure delay, all input changes will have an effect on output. In other words, outputs follows inputs after delay D. If the component has inertial delay, output will respond only to input changes which have persisted for time D. In other words, input pulses whose width is less than D will be absorbed by the component. This reflects the fact that short pulses contain insufficient energy to trigger a state change in a real component.

Sometimes, the delay of a component has a more general form. There may exist a threshold $T < D$ such that the component absorbs input pulses whose width is less than T. However output follows input if the pulse width is more than T. In DILL this is termed *general delay*. In fact, it could be considered as inertial delay T cascaded with a pure delay $D - T$. Figure 3 shows how inputs are related to outputs for different delay types. The scale of the figure takes T as 2 and D as 4 time units.

4.2 Delay Elements in DILL

The DILL library include the following delay elements that act like pseudo-components. Unlike fixed delays, all delays have a non-deterministic range from *MinDel* (the minimum delay) to *MaxDel* (the maximum delay). For general delay, *MinWidth* corresponds to the threshold T in the last section. It is obvious that the assumption of non-deterministic delays is more realistic and flexible than that of fixed delays. Besides the basic delay types that follow, Timed DILL also handles more complex aspects such as high-to-low, low-to-high and pin-to-pin delays [5].

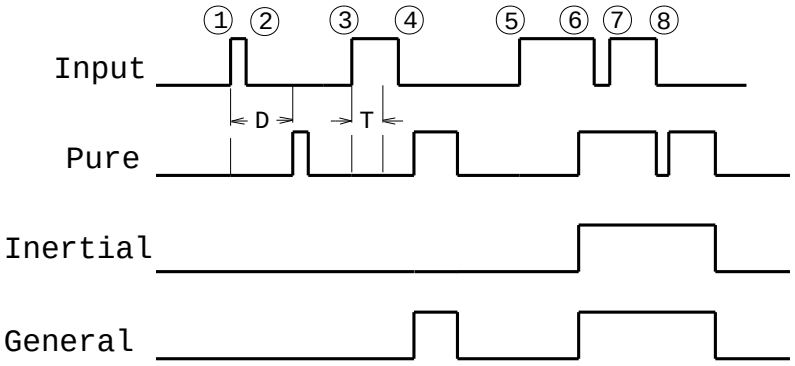

Fig. 3. Basic Delay Types

Inertial Delay A naive attempt at specifying inertial delay would use the ET-LOTOS generalised life reducer. If the interval between two input transitions is less than the delay, output must not occur. The exact delay would be determined by the environment because the delay range is associated with an observable action. But in DILL what should really be specified is that the delay is decided by the component itself. If the delay is connected to other components in a larger design, an output port might well be hidden. This would mean that the delay time is exactly *MinDel* instead of being a non-deterministic value since ET-LOTOS adopts maximal progress for hidden events. A better specification of inertial delay is given by:

```
process DelayInertial [Ip, Op] (MinDel, MaxDel: Time) : noexit :=
  DelayInertialAux [Ip, Op] (MinDel, MaxDel, 0 of Bit, 0 of Bit)
where
  process DelayInertialAux [Ip, Op]                    (* auxiliary definition *)
   (MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
    Ip ? NewDataIp : Bit;                                  (* input change *)
    DelayInertialAux [Ip, Op] (MinDel, MaxDel, NewDataIp, DataOp)
  []                                                              (* or ... *)
    [DataIp ne DataOp] ->                          (* output must change? *)
      i {MinDel, MaxDel};                            (* allow delay to pass *)
      Op ! DataIp {0};                          (* output changes at once *)
      DelayInertialAux [Ip, Op] (MinDel, MaxDel, DataIp, DataIp)
  endproc (* DelayInertialAux *)
endproc (* DelayInertial *)
```

The specification takes advantage of internal events. The internal event **i** introduces non-deterministic delay, which means the output port can change its value at any time between *MinDel* and *MaxDel*. The exact delay value is determined by the component itself and not by the environment. Moreover even if the component is connected to other components, the delay is still non-deterministic since only hidden events are urgent.

Pure Delay A pure delay may be specified with:

```
process DelayPure[Ip, Op] (MinDel, MaxDel : Time) : noexit :=
    DelayPureAux [Ip, Op] (MinDel, MaxDel, 0 of Bit, 0 of Bit)
  where
    process DelayPureAux [Ip, Op]                          (* auxiliary definition *)
     (MinDel, MaxDel : Time, DataIp, DataOp : Bit) : noexit :=
      Ip ? NewDataIp : Bit;                                (* input change *)
      (
        [NewDataIp eq DataOp] ->                           (* no output change? *)
          DelayPureAux [Ip, Op] (MinDel, MaxDel, NewDataIp, DataOp)
        []                                                 (* or ... *)
        [NewDataIp ne DataOp] ->                           (* output must change? *)
          (
            i {MinDel, MaxDel};                            (* allow delay to pass *)
            Op ! NewDataIp {0};                            (* output changes at once *)
            stop                                           (* delay behaviour now done *)
          |||                                              (* interleaved with ... *)
            DelayPureAux [Ip, Op] (MinDel, MaxDel, NewDataIp, NewDataIp)
          )
      )
    endproc (* DelayPureAux *)
endproc (* DelayPure *)
```

Because delay is assumed to be non-deterministic rather than fixed, the pure delay above may output 'strange' sequences like *Op ! 0; Op ! 0; Op ! 1; . . .* In this sequence, the second *Op ! 0* overtakes *Op ! 1* and results in the two consecutive *Op ! 0* events. The phenomenon of *catch-up* arises if a later input change takes less time to reach the output than an earlier input change. Figure 4 illustrates the reason for the phenomenon, assuming that the delay is between 3 and 9 time units. If both events *Op ! 0* and *Op ! 1* happen within the overlapped region then catch up can happen. Suppose the width of an input pulse is W. A necessary condition for catch-up is $W \leq MaxDel - MinDel$.

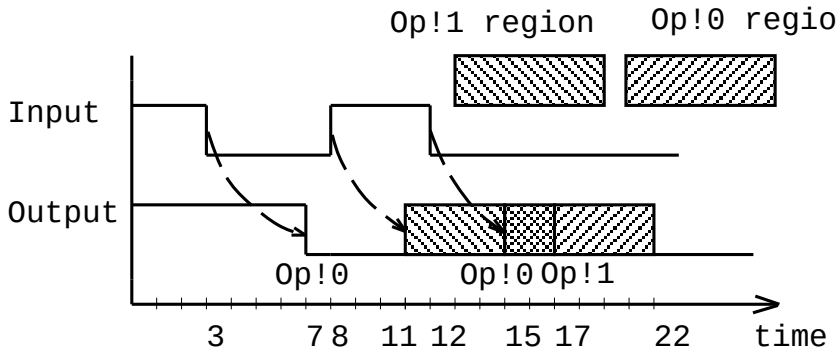

Fig. 4. Catch-Up Phenomenon with Pure Delay

Catch-up may exhibit various forms in real hardware if delays vary significantly. However, digital components generally operate in a stable environment so the variation in delays is in a narrow range. Thus the catch-up condition is rarely met in practice. The phenomenon exists in any delay model that is based on pure delay (e.g. the general delay component to be discussed soon). But it does not appear in the inertial delay model since an input change will prevent any pending output; it is therefore not possible to catch up a pending output.

General Delay As mentioned before, general delay has a threshold *MinWidth*. Input pulses whose width is less than *MinWidth* will be absorbed by the component. They will appear at the output if their width is greater than or equal to *MinWidth*. The general delay element in DILL is specified such that it can model not only a general delay but also inertial or pure delay. This is achieved by choosing appropriate timing parameters. The specification of the general delay will not be given in detail because it is just the combination of inertial and pure delay. The following gives the rules of using the timing parameters. Here *Inf* is the maximal value of the time domain (taken as arbitrarily large):

$0 < MinWidth < MinDel \leq MaxDel < Inf$ This describes general delay. The general delay model meaningful only when *MinWidth* is a positive number less than *MinDel*.

$MinWidth = 0, MinDel \leq MaxDel < Inf$ This is pure delay. The difference between a pure and a general delay is that for pure delay, *MinWidth* is zero so the component does not absorb a narrow pulse.

$0 \leq MinDel \leq MaxDel < Inf, MinWidth > MinDel$ This is inertial delay. It applies if the threshold *MinWidth* is greater than *MinDel*. *MinWidth* is often set to *Inf* for inertial delay.

$MinDel = 0, MaxDel = Inf, MinWidth > 0$ This is equivalent to an untimed delay component. Usually *MinWidth* is given the value *Inf*.

5 Timing Constraint Components

Timing constraints in DILL are used to check if the inputs of a component satisfy some conditions. Common timing constraint 'components' have been added to the DILL library, including those for setup, hold, pulse width and period.

Setup and hold times are always associated with flip-flops. For a D flip-flop, setup time is the time interval between a change in input *D* and the trigger that stores this data (e.g. a positive-going transition of the clock *Ck*). The data signal must then remain stable for a minimum time interval if correct operation of the flip-flop is to be guaranteed. For a flip-flop, the hold time is the interval in which input data must remain unchanged after triggering by the clock. Again, this minimum must be respected for correct operation. A timing diagram showing setup time and hold time is given in figure 5.

The setup time constraint is specified as follows, assuming that the active clock transition is positive-going. A similar approach is used to specify a hold time constraint, to check the minimum input pulse width, and to check the period of clock signals.

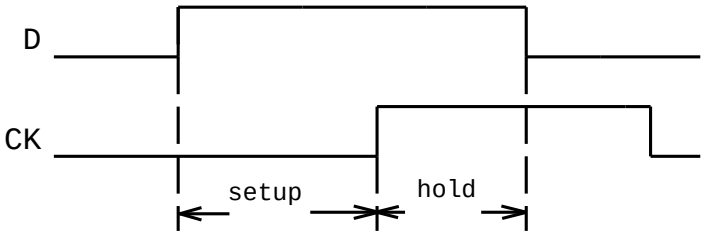

Fig. 5. Setup and Hold Times for D Flip-Flop

```
process SetupDel [D, Ck, Err] (SetupTime : Time) : noexit :=
   D ? NewDataIp: Bit;                                  (* new data input *)
   AfterD [D, Ck, Err] (SetupTime, SetupTime)           (* check setup time *)
[]                                                      (* or ... *)
   Ck ? NewClock : Bit;                                 (* new clock input *)
   SetupDel [D, Ck, Err] (SetupTime)                    (* no setup time to check *)
endproc (* SetupDel *)

process AfterD [D, Ck, Err] (SetupTime, SetupRem : Time) : noexit :=
   delta (SetupRem) i;                                  (* enforce min. setup time *)
   SetupDel [D, Ck, Err] (SetupTime)                    (* restart setup check *)
[]                                                      (* or ... *)
   Ck ? NewClock : Bit @ t;                             (* new clock input *)
   (
      [NewClock eq 0] ->                                (* negative-going clock? *)
         AfterD [D, Ck] (SetupTime, SetupTime - t)      (* check setup time left *)
   []                                                   (* or ... *)
      [NewClock eq 1] ->                                (* positive-going clock *)
         Err ! SetupError;                              (* min. setup time violated *)
         SetupDel [D, Ck, Err] (SetupTime)              (* restart setup check *)
   )
[]                                                      (* or ... *)
   D ? NewDataIp: Bit;                                  (* new data input *)
   AfterD [D, Ck, Err] (SetupTime, SetupTime)           (* restart setup check *)
endproc (* AfterD *)
```

6 Timed DILL Example: 2-to-1 Multiplexer

As an example, a 2-to-1 multiplexer will be specified and analysed. This component has two data inputs *A* and *B*, a selection input *S* and an output *C*. A selection input of 0 or 1 chooses input *A* or *B*, which appears at *C* after some delay. The delays used in the example are inertial, mainly because they are easy to handle but are general enough to represent delay in most digital circuits.

The multiplexer is specified at two levels. The higher level specifies the required behaviour and timing performance. The lower level specifies the structure of the component by connecting basic logic gates. The lower level implements the higher level. The timed specifications were analysed through simulation and testing.

6.1 Behavioural Specification

The behavioural specification of the 2-to-1 multiplexer in DILL is as follows:

```
define(MinDel, 10)                                  # min. delay value
define(MaxDel, 15)                                  # max. delay value
include(dill.m4)                                    # include DILL library
circuit(                                            # circuit description
   Multiplexer2to1_BB [A, B, S, C],                 # circuit name and ports
   hide InC in                                      # internal gate to delay
      Multiplexer2to1_BB_0 [A, B, S, InC]           # multiplexer instance
   |[InC]|                                          # sync with delay
      Delay [InC, C] (Inf, MinDel, MaxDel)          # delay instance
   where
      Multiplexer2to1_BB_0_Decl                     # multiplexer from library
)
```

DILL provides a veneer on top of LOTOS – mainly a library of components that can be combined using LOTOS operators. The **circuit** declaration names the overall specification and its parameters. It then gives a LOTOS behaviour expression for the whole circuit. Library components are declared and automatically included by giving their names (*Component_Decl*). In the above, *Multiplexer2to1_BB_0* is a 2-to-1 multiplexer in behavioural style (*BB* = black box) that exhibits zero delay (*0*). It was derived from the corresponding component using the approach in section 3. The behavioural specification defines an inertial delay between *MinDel* (10) and *MaxDel* (15).

The behavioural specification was validated using the TE-LOLA step-by-step simulator. Basically, the behaviour of the multiplexer is simulated for each input combination to see if it is as expected. The results of simulation are regarded as the criteria against which simulation of the lower-level specification should be judged.

Because test results from TE-LOLA become inconclusive if tests have actions with intervals such as **i** $\{5,7\}$, it is possible to use only single delay values in the tests themselves. For a higher-level specification like the behavioural one, such an assumption would be unrealistic. There are several paths from the inputs to a output, so the delay associated with the output has a range of values.

6.2 Structural Specification

The structure of the 2-to-1 multiplexer is shown in figure 6. The logic gates in the diagram are timed gates. Each of them consists of zero-delay logic and a delay component. The inset in the figure shows the structure of the *and* gate *G2*; *0_D* in the figure means zero delay. Other gates have the same kind of structure. The design of the multiplexer

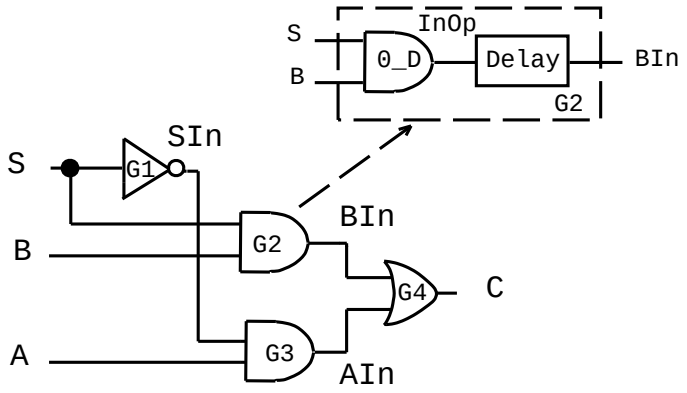

Fig. 6. Structure of 2-to-1 Multiplexer as Timed Logic Gates

can be found in a standard textbook. However, as will be seen later this design contains hazards. Assuming the delay of each gate is 5 time units, the corresponding DILL specification is:

```
define(DelayData, Inf, 5, 5)                    # MinWidth, MinDel, MaxDel
include(dill.m4)                                  # include DILL library
circuit(                                          # circuit description
  timed,                                          # declare timed design
  Multiplexer2to1 [A, B, S, C],                   # circuit name and ports
  hide AIn, BIn, SIn in                           # internal gates
    Inverter [S, SIn]                             # inverter instance
  ||[S, SIn]||                                    # sync with selection signals
    (
      And2 [SIn, A, AIn]                          # two-input and instance
    |||
      And2 [S, B, BIn]                            # two-input and instance
    )
  ||[AIn, BIn]||                                  # sync with inputs
    Or2 [AIn, BIn, C]                             # two-input or instance
  where
    Inverter_Decl                                 # inverter from library
    And2_Decl                                     # two-input and from library
    Or2_Decl                                      # two-input or from library
)
```

This specification first defines the delay of the basic logic gates. Here the delay is fixed at 5 and is inertial because *MinWidth* is *Inf*. The first parameter of the **circuit** declaration is optional. In this example it is **timed**, indicating that basic logic gates use the delay data defined by the specifier. The logic family name (e.g. CMOS) may also be given, as delays for such families are pre-defined in Timed DILL. The default value

for a circuit is **untimed**, which appends *Inf, 0, Inf* to every instantiation of a basic logic gate. For a higher-level specification like the one in section 6.1 there is no need to define the delay parameters because basic logic gates are not used. But this does not means that the component is untimed.

Timed behaviour was investigated using the *TestExpand* function of TE-LOLA that automatically explores a test in parallel with a specification. Testing was done by composing test processes in parallel with the specification. If the test process can be followed for all executions of the composed specification, the result of testing is *must pass*. If the test process can be followed only for some executions, the result is *may pass*. Otherwise the test is considered to be *rejected*.

Firstly, the functionality of the multiplexer was tested. Test processes were defined to check if the output correctly corresponds to each possible input combination. For example, for *A=1, B=1, S=0* the output should be *C=1* after 10 or 15 time units (because the input-output paths contain 2 or 3 levels of basic gate). Thus the corresponding test process is as follows. The test successfully passed as expected. The other input combinations were tested in a similar way.

```
process Test110_1 [A, B, S, C, OK] : noexit :=      (* inputs 110, output 1 *)
  A ! 1 {0}; B ! 1 {0}; S ! 0 {0};                    (* accept inputs *)
  (
    C ! 1 @ t [t = 10]; OK; stop                      (* output at time 10 *)
  []                                                  (* or ... *)
    C ! 1 @ t [t = 15]; OK; stop                      (* output at time 15 *)
  )
endproc (* Test110_1 *)
```

Secondly, there were tests to see if the design had a timing hazard. Hazards are unwanted transitions that appear on the outputs of digital circuits in response to the changes on inputs. For example, suppose that the output should stay the same (e.g. *1*) after the input state changes from I_1 to I_2. However, in an actual implementation the output may change from *1* to *0* and back again after an input transition. The consecutive unwanted transitions *1* to *0* and *0* to *1* are hazards. Figure 7 illustrates kinds of common hazards in circuits and their corresponding LOTOS specifications. Cases (a) and (b) are called *static* hazards, while (c) and (d) are called *dynamic* hazards. The 1-0-1 example corresponds to (b) in the figure.

(a)	(b)	(c)	(d)
`Op ! 1; Op ! 0`	`Op ! 0; Op ! 1`	`Op ! 0; Op ! 1`	`Op ! 1; Op ! 0`

Fig. 7. Hazards and their LOTOS Specifications

Transition	Type of Hazard	Changed Inputs
000 to 101	static 0	2
010 to 101	static 0	3
011 to 100	static 1	3
011 to 110	static 1	2
111 to 100	static 1	2
111 to 110	static 1	1

Fig. 8. Hazards in the 2-to-1 Multiplexer

The multiplexer has 3 input ports and thus 8 input states. Each input state may change to one of the other 7 input states, so there are 56 possible input transitions (8×7) in total. These transitions were checked with tests that deliberately risked hazards. If the design of the multiplexer is hazard-free, then the hazard tests should be rejected. Unfortunately 6 of the 56 transitions pass the tests, i.e. they exhibit hazards. Figure 8 lists these transitions and the corresponding hazards. The test results indicate that when the delays of each gates are fixed, the circuit exhibits static hazards. One of the hazards happens when there is a single input change; the others occur when more than one input changes simultaneously.

The test corresponding to the transition from *ABS* 111 to 110 looks like:

```
process Test111_110Hazard [A, B, S, C, OK] : noexit :=(* inputs 110 to 110 *)
  (
    A ! 1 {0}; B ! 1 {0}; S ! 1 {0};                    (* accept inputs *)
    (
      C ! 1 @ t [t = 10]; exit                          (* output at time 10 *)
    []                                                  (* or ... *)
      C ! 1 @ t [t = 15]; exit                          (* output at time 15 *)
    )
  )                                                     (* now state 111 *)
 >>                                                     (* continue with ... *)
  (
    S ! 0 @ t [t = 2];                                  (* input change, state 110 *)
    (
      C ! 0 @ t [t = 10];                               (* hazard *)
      C ! 1 @ t [t = 5]; OK; stop
    []                                                  (* or ... *)
      C ! 0 @ t [t = 15];                               (* hazard *)
      C ! 1 @ t [t = 5]; OK; stop
    )
  )
endproc (* Test111_110Hazard *)
```

The output transitions *C ! 0* and *C ! 1* in the process indicate a hazard because the output should remain at 1 for the transition 111 to 110. By simulating a passed test

sequence, the reason for the hazards is discovered: the inputs follow different lengths of path to reach the output. Figure 9 is a convenient solution, although perhaps not the best one. Three redundant repeaters are used to guarantee that each input-output path has exactly three gate delays. It is obvious that the delay of each repeater should also be 5 time units. The total delay of the new design is 15 time units, which complies with the timing constraint expressed by the behavioural specification.

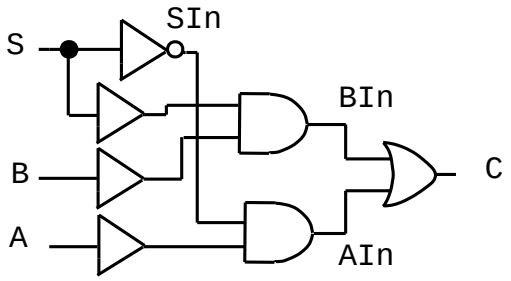

Fig. 9. The Hazard-Free Multiplexer

7 Conclusion

DILL allows formal specification and analysis of digital hardware. It has extended the experience with LOTOS in the communications field. Timed DILL offers a number of important benefits. It can check whether timing requirements are respected by a design, making use of timing constraint components. Potential timing errors like hazards can be discovered, as in the multiplexer example. Timed DILL can also be used to analyse performance such as minimum/maximum delays and timing on critical paths. Although the paper has deliberately been illustrated with only a small example, the approach is applicable to much larger problems. A future goal is support of Timed DILL with verification based on KRONOS, HYTECH or timed automata.

References

1. 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.
2. Ji He. *Formal Specification and Analysis of Digital Hardware Circuits in LOTOS*. PhD thesis, Department of Computing Science and Mathematics, University of Stirling, UK, April 2000.
3. 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, August 2000.
4. Ji He and Kenneth J. Turner. Extended DILL: Digital logic with LOTOS. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, November 1997.

5. Ji He and Kenneth J. Turner. Timed DILL: Digital logic with LOTOS. Technical Report CSM-145, Department of Computing Science and Mathematics, University of Stirling, UK, April 1998.
6. Ji He and Kenneth J. Turner. Modelling and verifying synchronous circuits in DILL. Technical Report CSM-152, Department of Computing Science and Mathematics, University of Stirling, UK, April 1999.
7. Ji He and Kenneth J. Turner. Protocol-inspired hardware testing. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Proc. Testing Communicating Systems XII*, pages 131–147, London, UK, September 1999. Kluwer Academic Publishers.
8. Ji He and Kenneth J. Turner. Specification and verification of synchronous hardware using LOTOS. In Jianping Wu, Samuel T. Chanson, and Quiang Gao, editors, *Proc. Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)*, pages 295–312, London, UK, October 1999. Kluwer Academic Publishers.
9. Luc Léonard and Guy Leduc. An enhanced version of timed LOTOS and its application to a case study. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 483–500. North-Holland, Amsterdam, Netherlands, 1994.
10. Luis Llana and Gualberto Rabay Filho. Defining equivalences between time/action graphs and timed action graphs. Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, December 1995.
11. Santiago Pavón Gomez, David Larrabeiti, and Gualberto Rabay Filho. LOLA user manual (version 3R6). Technical report, Department of Telematic Systems Engineering, Polytechnic University of Madrid, Spain, February 1995.
12. Gualberto Rabay Filho and Juan Quemada. TE-LOLA: A timed LOLA prototype. In Zmago Brezocnik and Tatjana Kapus, editors, *Proc. COST 247 International Workshop on Applied Formal Methods*, pages 85–95, Slovenia, June 1996. University of Maribor.
13. Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyar, editors, *Proc. Formal Description Techniques VI*, pages 71–86, Amsterdam, Netherlands, 1994. North-Holland.