Specifying Hardware Timing with ET-LOTOS

Ji He and Kenneth J. Turner

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

Abstract. It is explained how DILL (Digital Logic in LOTOS) can specify and analyse hardware timing characteristics using ET-LOTOS (Enhanced Timed LOTOS – the ISO Language Of Temporal Ordering Specification). Hardware functionality and timing characteristics are rigorously specified and then validated.

1 Introduction

DILL (Digital Logic in LOTOS [2]) is an approach for specifying digital circuits using LOTOS (Language Of Temporal Ordering Specification [1]). 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.

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 paper uses ET-LOTOS (Enhanced Timed LOTOS [3]). Because ET-LOTOS tools are currently under development, the authors have also used TE-LOTOS (Time Extended LOTOS [5]). Although these LOTOS variants adopt different semantic models, the equivalence between them has been established [4].

ET-LOTOS is a timed LOTOS that allows the modelling of time-sensitive behaviour. The delay operator *delta (time)* means that the subsequent behaviour will be delayed by *time*. A time value is relative to the instant when the previous action occurs. The time measurement operator $@t$ is used to measure the time elapsed between the instant when an event is 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. 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, i.e. if a hidden action can occur it must happen at once (unless an alternative action occurs).

The input-output timing relationship is normally called *delay*. A timing relationship among inputs is called a *timing constraint*, meaning that the digital circuit can work correctly only if the constraints are met. In an *integrated method* for describing delay,

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 161–166, 2001.

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. Untimed behaviour should also be a special case of timed behaviour. *Combined methods* for specifying delay are thus preferred. These separate the functionality and the timing characteristics into different processes.

The approach selected for Timed DILL is called the *parallel-serial* model. Functionality is assumed to be specified with zero delay. Timing constraints are placed in parallel with the inputs of the functional specification to check if input requirements are met. Delays are placed in series with outputs of the functional specification to describe overall delay. Error events are introduced to discover violation of timing constraints; they have no counterpart in a real physical component.

If the timing constraints are void and the delays are between zero and arbitrarily large, the timed model is equivalent to the untimed model. An untimed specification is thus just a special case. Component functionality is supposed to have zero delay. This can be easily obtained from the untimed functionality. To change an untimed specification to one with zero delay, a life reducer $\{0\}$ is appended to each output event offer.

2 Delays and Timing Constraints

Suppose the delay of a digital component is D. If a component has *pure delay*, all input changes will have an effect on output. If a component has *inertial delay*, output will respond only to input changes that have persisted for time D. 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 can be considered as inertial delay T cascaded with a pure delay $D - T$.

The DILL library supports non-deterministic delays ranging from *MinDel* (minimum) to *MaxDel* (maximum). For general delay, *MinWidth* corresponds to the threshold T. Timed DILL also handles high-to-low, low-to-high and pin-to-pin delays.

A naive attempt at specifying inertial delay would use the ET-LOTOS generalised life reducer. 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 internal event **i** introduces non-deterministic delay, i.e. output can change at any time between *MinDel* and *MaxDel*. The exact delay 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.

Because delay is assumed non-deterministic rather than fixed, the pure delay specified below exhibits sequences like *Op ! 0; Op ! 0; Op ! 1* where the second *Op ! 0* overtakes *Op ! 1* and results in 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. 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.

```
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 *)
```

The general delay element in DILL is specified such that it can model not only a general delay but also inertial or pure delay by choosing appropriate timing parameters. The specification of general delay is not given here as it is just the combination of inertial and pure delay. The following gives the rules of using the timing parameters. *Inf* is the maximal value of the time domain (taken as arbitrarily large):

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

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

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

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

Timing constraints in DILL are used to check if the inputs of a component satisfy some conditions. Common timing constraint elements 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. Setup time is the time interval between a change in data input and the trigger that stores this data. The hold time is the interval in which input data must remain unchanged after triggering by the clock. The setup time constraint is specified as follows for a D flip-flop where the active clock transition is positive-going. A similar approach specifies a hold time constraint, checks the minimum input pulse width, or checks the period of clock signals.

```
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 *)
```

3 Timed DILL Example: 2-to-1 Multiplexer

As an example, a 2-to-1 multiplexer will be specified and analysed. A selection input S of 0 or 1 chooses input A or B, which appears at C after some delay. A higher level specification defines the required behaviour and timing performance. A lower level specification gives a component design that implements the higher level. 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*).

The behavioural specification was validated using the TE-LOLA 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. The design of the 2-to-1 multiplexer uses the selection signal to gate one or other input. Although this design might be found in a standard textbook, it was found that it contains hazards. 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
```

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**; the default value is **untimed**, which appends *Inf, 0, Inf* to every instantiation of a basic logic gate.

Timed behaviour was investigated using the *TestExpand* function of TE-LOLA that automatically explores a test in parallel with a 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. Secondly, there were tests to see if the design had a timing hazard (static or dynamic). Input transitions were checked with tests that deliberately risked hazards. Unfortunately 6 of the 56 transitions pass the tests, i.e. they exhibit hazards when the delays of each gates are fixed (transitions 000→101, 010→101, 011→100, 011→110, 111→100, 111→110). By simulating a passed test sequence, it becomes obvious that hazards are due to inputs following different lengths of path to reach the output. One solution is to introduce delay elements to equalise input-output path lengths.

4 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 and Kenneth J. Turner. DILL (Digital Logic in LOTOS) project web page. http://www.cs.stir.ac.uk/∼kjt/research/dill.html, November 2000.
3. 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.
4. 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.
5. 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.