Technical Report

UCAM-CL-TR-122
ISSN 1476-2986

Number 122

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Temporal abstraction of digital designs

John Herbert

February 1988

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Temporal Abstraction of Digital Designs

John Herbert
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG

Abstract: Formal techniques have been used to verify the function of reasonably large digital devices ([Hunt85], [Cohn87]), and also to describe and reason about digital signal behaviour at a detailed timing level [Hanna85] [Herbert86]. Different models are used: simple synchronous models of components are the basis for verifying high-level functional specifications; more detailed models which capture the behaviour of signals in real time are the basis for proofs about timing. The procedure called *temporal abstraction* is a technique for formally relating these two behavioral models.

The background to temporal abstraction is presented and the details of its implementation in HOL. The HOL language ([Gordon85a]) is a computerised version of higher-order logic which has an associated proof assistant also called HOL. In HOL one may specify behaviour at both the functional and timing levels. This work describes how the relationship between these levels may also be described in HOL and reasoned about using the HOL system.

The formal transformation of descriptions of behaviour at the timing level to behaviour at the functional level involves generating and verifying timing constraints. This process can be identified with the conventional design activity of timing analysis. This work shows that timing verification can be viewed, not as a separate phase of design, but as part of a single verification process which encompasses functional and timing verification. A single formal language, HOL, is used to describe all aspects of the behaviour and a single verification system provides all the proofs of correctness. The use of uniform, formal techniques is shown to have a number of advantages.

Preface

This report is essentially a self-contained portion of my thesis [1]. The following is a copy of my acknowledgement from the thesis. In addition I would like to thank Dr. Mike Gordon and Dr. Mary Sheeran for their comments.

Part of the work was supported by SERC/Alvey grant number GR/D/17304 entitled "Formal Methods for Hardware Verification".

I wish to thank my supervisor, Dr. Andy Hopper, for his help and support. I am grateful to Professor Roger Needham, the head of the Computer Laboratory, for the opportunity to work at Cambridge and for his contribution to a good working environment. I have been fortunate to receive financial support through scholarships from the Royal Commission for the Exhibition of 1851 and Corpus Christi College, Cambridge, and an award from the Lundgren Research Fund.

I would like to thank those people who have read and commented on earlier drafts of this dissertation: Andy Hopper, Jeff Joyce, Miriam Leeser, Tom Melham, Larry Paulson and Frances Quigg. Frances Quigg read early drafts diligently and helped correct my presentation of ideas; Miriam Leeser commented closely on the chapters on timing.

The hardware verification group at Cambridge has provided a friendly and stimulating work environment. I am especially grateful to Mike Gordon for leading the way in hardware verification and giving me advice and encouragement when it was needed. Special thanks also to the other HVG roadshow members — Albert, Inder, Miriam and Tom.

[1] Application of Formal Methods to Digital System Design, J. M. J. Herbert, Ph D Thesis University of Cambridge, 1986

Contents

1 Introduction

All the formal specifications and proofs described in this article have been constructed using the HOL system. We will not describe the HOL language and associated system; they have been presented in [Gordon85a], [Gordon85b].

In previous work we have reasoned in HOL about digital systems at two distinct behavioural levels. The HOL ECL chip proof, [Herbert88a], deals with the synchronous behavioural level; each combinational device has zero delay, a memory element acts as a unit delay. In the verification of basic memory devices,[Herbert88b], we have reasoned about digital behaviour at the timing level; each gate has propagation delay, each memory element has an explicit clock and a number of timing parameters. We now relate the behaviour of the timing level models to the corresponding synchronous level models.

The process of formally relating behaviour on different time scales is called *temporal abstraction.* (Note, we sometimes refer to temporal abstraction as simply abstraction.) In this work temporal abstraction is used to relate the timing level to the synchronous level. We derive a formal relationship between timing level models which capture closely the physical component behaviour, and synchronous level models which provide a tractable means of verifying functional behaviour. Related work on various abstraction mechanisms (including temporal abstraction) is being done by Melham [Melham87].

Before presenting a formal treatment of temporal abstraction we describe some of the ideas behind this process.

1.1 Informal Description of Abstraction

At the timing level, the clock signal is modelled just like any other signal. At the synchronous level, the clock signal is hidden. However, we can consider the time dimension at the synchronous behavioural level to be based on the times of synchronising events of the hidden clock (*e.g.* the times of rising edges for positive-edge triggered logic). This relationship provides the basis for relating the time dimension of the two levels. We identify a signal at the synchronous level with a sampled timing level signal. Successive synchronous signal values are the values of the corresponding timing level signal, sampled on successive clocking events.

We have chosen to clarify the abstraction process by only sampling at the exact time of the active clock event and by sampling all signals at the same time. An identical process can be used when signals are sampled at some offset from clock

event times and when different signals are abstracted in different ways.

Figure 1 presents a timing diagram for signals of a positive-edge triggered flip-flop. The value of data input d around the active edge is latched by the flip-flop and appears on the q output some time after the active edge.

We sample the data input and output signals at the times of rising edges of the clock. The sampled value of q at times t0 + 1 and t0 + 2 are the sampled value of d at time t0 and t0 + 1 respectively. For these waveforms, the flip-flop acts as a unit delay between the sampled d and sampled q signals. This corresponds to our synchronous level model of flip-flop behaviour. A rigorous proof of this relationship will be given later.

Sampling the timing level signals is not sufficient to deduce that the synchronous level behaviour is exhibited. A number of timing conditions must be satisfied by the timing level signals. For example, if a timing level flip-flop is to exhibit the behaviour of a unit delay at the synchronous level, then its data and clock inputs must fulfill setup, hold and minimum high and low constraints. Dealing efficiently with timing conditions is an important part of the verification of digital designs modelled at the timing level.

2 Formalising Clock Properties

Certain concepts about clocks must be formalised for the temporal abstraction process. Although our particular abstraction deals with regular periodic clocks, many of the properties we formalise do not depend on periodicity. A similar set of definitions and theorems deals with rising and falling edges. We just present those for rising edges.

2.1 Rising Edges

We model a signal as a function from time to boolean. At each moment in time a signal may have a value of T or F. A signal rise, which corresponds to the change from value F to T, is thought of intuitively as occurring between two instants of time. Formally, we define a signal as rising at time t if it is low at time t and high at time t+1.

$$\texttt{RISE (signal,t)} = \neg\,\texttt{(signal(t))} \wedge \texttt{signal(t+1)}$$

The formal rise time can be brought within a given limit of the intuitive transition time by choosing a sufficiently fine time scale. (This assumes that signals are

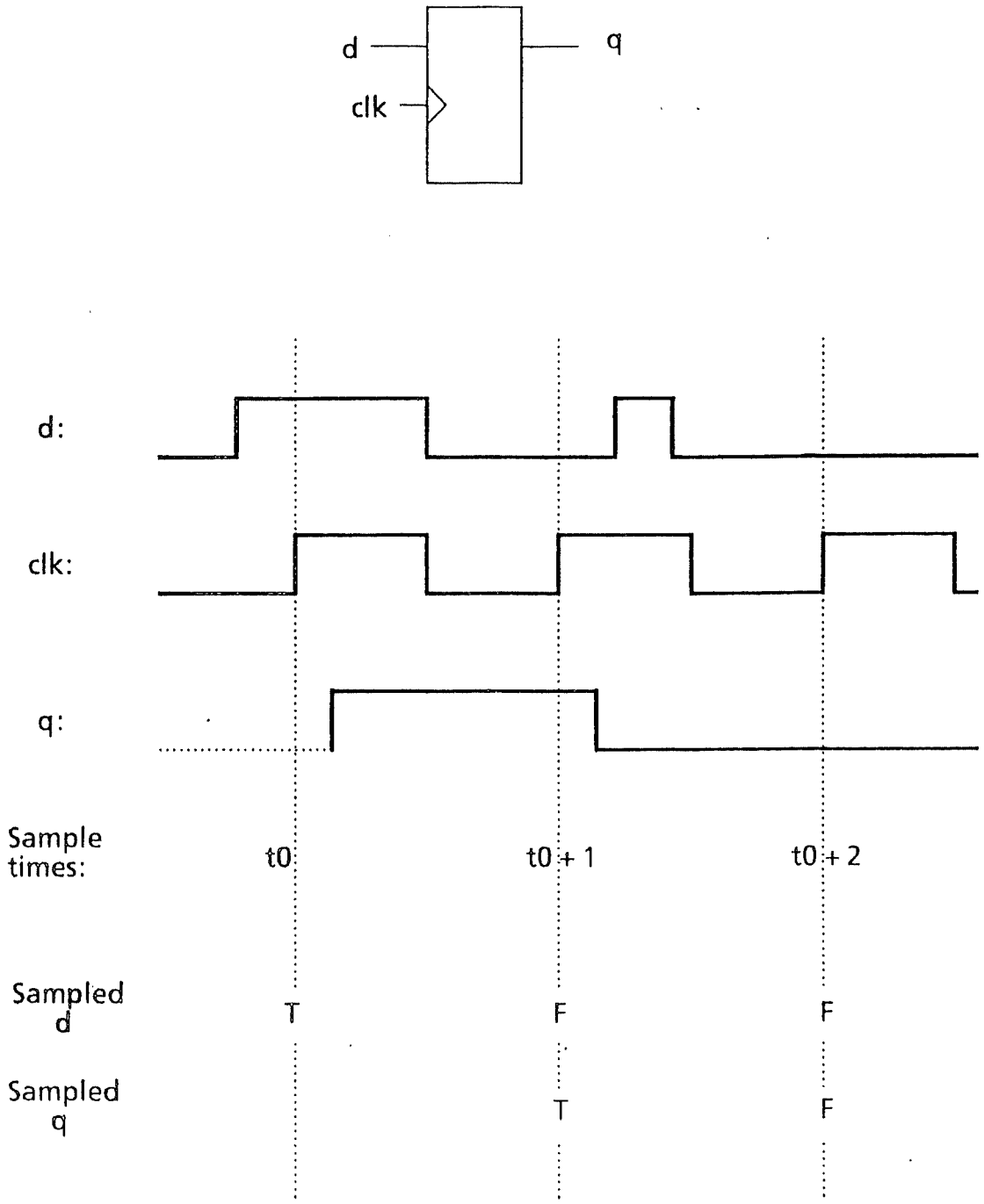

Figure 1: Sampling of flip-flop signals

two-valued.)

A clock signal provides a succession of edges. We wish to introduce the concept of the n^{th} rise of the clock. To do this we define a higher order function UP using primitive recursion:

```
(∀ ck t.
 (UP 0 ck t) = (RISE(ck,t) ∧ (∀ t1.(t1 < t) ⟹ ¬ (RISE(ck,t1))))) ∧
(∀ n ck t.
 (UP (SUC n) ck t) =
              (RISE(ck,t)   ∧
              (∃ t1.(t1 < t)   ∧
              (∀ t0.((t1 < t0) ∧ (t0 < t)) ⟹ ¬ (RISE(ck,t0))) ∧
              (UP n ck t1))))
```

This definition can be paraphrased as:

> the zeroth UP of the clock is at time t if the clock rises at time t and does not rise before then; the (n+1)th UP of the clock is at time t if the clock rises at time t and there exists a time t1 less than t such that the nth rise of the clock is at time t1 and the clock does not rise between t and t1.

2.2 Regular Signal Rises

We now introduce the idea of a rise cycle which consists of two successive rising edges.

```
CLOCK_RISE_CYCLE(ck, t1, t2)
     = ((t1 < t2) ∧ RISE(ck,t1) ∧ RISE(ck,t2) ∧
       (∀ t.((t1 < t) ∧ (t < t2)) ⟹ ¬ (RISE(ck,t))))
```

Times t1 and t2 define a single rise cycle of the clock if t1 and t2 are successive rise times.

A regular synchronous clock will include a periodic train of rising edges. A further definition introduces the concept of a periodic rising edge.

```
CK_RISE_PERIODIC(ck,origin,p) =
    (∀ n.CLOCK_RISE_CYCLE(ck,(n*p)+origin,((n+1)*p)+origin)) ∧
    (∀ t.(t < origin) ⟹ ¬ RISE(ck,t)) ∧
    ¬ (p=0)
```

The predicate CK_RISE_PERIODIC is true if the signal ck has a periodic train of rising edges beginning at time origin and the period p is non-zero.

2.3 Time of n^{th} UP

We deduce that the n^{th} UP of a clock ck, for which CK_RISE_PERIODIC(ck,origin,p) holds, occurs when the time is (n*p)+origin. The theorem is:

```
CK_RISE_PERIODIC(ck,origin,p) ⟹
   (∀n t.UP n ck t = (t=(n*p)+origin))
```

(The n^{th} rising edge of a clock with period p and start-up time origin occurs at time (n*p)+origin.)

Although the result seems obvious we have proved it rather than just stated it as an axiom. This arises from a philosophy of building from a reasonably small conceptual basis. We have tried to restrict the number and complexity of the definitions and axioms used. For example, the definition of the n^{th} rising edge did not assume periodicity or anything about falling edges. Other than in the basic definition of RISE, we have not referred to signals having the values T or F.

In a similar manner, the corresponding theorems for falling edges can be proved. Higher-order functions FALL and DOWN are analogous to RISE and UP.

2.4 Periodic Rises and Falls

Based on the theorems already proved about sequences of rising and falling edges, we now formalise regular clock cycles where both rising and falling edges occur. A single cycle of a clock is defined in terms of rising and falling signals. A predicate ONE_CYCLE is defined as follows:

```
ONE_CYCLE(ck,t0,p_low,p_high) =
       FALL(ck,t0) ∧
       DURING(t0,t0+(p_low-1))(λt. ¬RISE(ck,t)) ∧
       RISE(ck,t0+p_low) ∧
       DURING(t0+p_low,t0+(p_low+(p_high-1)))(λt. ¬FALL(ck,t)) ∧
       FALL(ck,t0+(p_low+p_high))
```

The predicate ONE_CYCLE is true if a single cycle of clock ck, as described in terms of rising and falling edges on the left-hand side of the definition, occurs. We might expect that p_low and p_high correspond to the lengths of time when the clock is low and high respectively. However the definition was just based on the concepts of rises and falls and does not refer to the logic levels.

A periodic clock consists of a regular succession of cycles. We define a predicate CLOCK_CYCLES(ck,origin,space,mark) as follows:

```
CLOCK_CYCLES(ck,origin,space,mark) =
```

```
(∀ n.
  ONE_CYCLE(ck,(n*(space+mark))+origin,space,mark) ∧
  0 < space ∧
  0 < mark ∧
  (∀ t. t < origin ⟹ ¬ FALL(ck,t)) ∧
  (∀ t. t < (origin+space) ⟹ ¬ RISE(ck,t)))
```

The predicate CLOCK_CYCLES describes what we expect to be true for a periodic clock. The clock has regular cycles of rising and falling edges, non-zero intervals between edges and no irregular edges before the regular cycles begin.

Although we have not referred to logic levels in the definitions, we do need to deduce when the clock signal is high and low. We deduce the following theorem:

```
CLOCK_CYCLES(ck,origin,space,mark) ⟹
     (DURING
      ((n*(space+mark))+origin,
       (n*(space+mark))+(origin+space))
      (λ t. ¬ ck t) ∧
      DURING
      ((n*(space+mark))+(origin+space),
       ((n+1)*(space+mark))+origin)
      (λ t. ck t)) ∧
     DURING
     (((n+1)*(space+mark))+origin,
      (((n+1)*(space+mark))+origin)+space)
     (λ t. ¬ ck t)
```

Thus space and mark correspond to the lengths of time when the clock ck is low and high respectively. The clock period is space + mark and the offset from the origin is origin.

In later proofs, we need a particular form for the higher-order function which relates the n^{th} rising edge to its time of occurrence. UP does not match this format. To facilitate the later proofs we define a new function UP_OF. This change of higher-order function is essentially cosmetic.
UP_OF is defined as:

```
UP_OF clk = (λ n. UP n clk)
```

The predicate CLOCK_CYCLES describes regular cycles of rising and falling edges. We can easily deduce the behaviour of the train of rising edges. We then use the theorems proved earlier to deduce the times of rising clock edges:

```
CLOCK_CYCLES(ck,origin,space,mark) ⟹
   (∀ n t. UP_OF ck n t = (t=(n*(space+mark))+(origin+space)))
```

2.5 Stability With Respect To a Clock Signal

We require that some signals fulfill timing conditions with respect to the clock signal. Two higher-order functions which are used to describe these conditions are STABLE_INTERVAL and STABLE_ABOUT_UP.

STABLE_INTERVAL is defined as follows:

```
STABLE_INTERVAL(t0,t1)event sig =
    (∃ data. DURING(event - t0,event + t1)(λ t. sig t = data)) ∧
    0 < t0 ∧
    t0 ≤ event
```

STABLE_INTERVAL(t0,t1)event sig states that the signal sig has some constant value over the interval event - t0 to event + t1, and the conditions 0 < t0 and t0 ≤ event are met. The condition 0 < t0 means that sig must be defined at time event and the interval is non-empty. The condition t0 ≤ event means that the length of the interval is always t0 + t1.

STABLE_ABOUT_UP is defined as:

```
STABLE_ABOUT_UP(clk,startcycle)(t0,t1)sig =
    (∀ n. STABLE_INTERVAL(t0,t1)(ε t. UP_OF clk(n + startcycle)t)sig)
```

(ε is the choice operator in HOL.

ε t.P t is a value of t chosen such that P t is true.)

STABLE_ABOUT_UP(clk,startcycle)(t0,t1)sig states that for all n, the signal sig is stable around a time chosen such that clk has the n + startcycle rising edge at that time.

This corresponds to stating that the signal is stable around all rising edges of the clock after (and including) the startcycle rising edge.

3 Temporal Abstraction Basics

A synchronous level signal may be thought of as a sampled version of a corresponding timing level signal. We consider the value of the synchronous level signal at time n to be the n^{th} sampled value of the timing level signal. A selection function is used to determine whether a value at the timing value is sampled.

3.1 Definition of Abstraction Function

The function used to relate the timing level to the synchronous level is ABS.

ABS is defined by:

```
ABS select signal n = signal(ε t. select n t)
```

`ABS select signal n` (*i.e.* the value of `signal`, abstracted using the function `select`, at synchronous time step `n`) is defined to be the value of `signal` at a time `t` chosen such that `select n t` is true.
`select n t` is true if time `t` at the timing level corresponds to the synchronous time `n`. The function `select` determines which values at the timing level are sampled to get values at the synchronous level.
`ABS select signal` (which is a function from natural numbers to any type) is the synchronous level signal corresponding to `signal` at the timing level.

Although we have explained the abstraction function in terms of abstraction from the timing to the synchronous level, it can be used to relate any two description levels which can be characterised in a similar way.

We have chosen to model signals at the timing level as functions of the natural numbers, and a unit interval corresponds to some interval of real time. At the synchronous level, signals are also modelled as functions of natural numbers. A unit interval now corresponds to the intervals between synchronizing clock events. In relating these two levels, the sampling times are the times of synchronizing clock events. For example, the sample times will be times of rising clock edges for positive-edge triggered logic.

3.2 Theorems of Abstraction

We call a theorem which states a relationship between timing level behaviour and its counterpart synchronous behaviour a *theorem of abstraction.*

The essential part of such theorems is the statement of the relationship between the two levels of abstraction. Certain timing conditions must be fulfilled if the abstraction from the timing to the synchronous level is to hold. Some of these timing conditions relate to the stability of input signals. When composing two devices, one may need to verify stability conditions on the interface signals. To allow this verification, the stability of output signals is also deduced when the abstraction of a device is performed.
The theorems of abstraction we deduce, follow this general format:

```
input stability conditions  ⟹
       clocking conditions      ⟹
           timing level behaviour ⟹
             synchronous level behaviour  ∧
             output stability assertions
```

This may be paraphrased as:
IF

the input stability conditions are satisfied, and

the clocking conditions are satisfied, and

the timing level behaviour holds,

THEN

the synchronous level behaviour holds, and

certain output stability assertions hold.

This form of abstraction theorem allows us to compose devices easily. When composing, we can use the assertions of output stability of one device to eliminate the corresponding input stability conditions of another device.

4 Abstraction of Component Models

In this section we describe how we formally relate timing level component models to their synchronous level counterparts.

4.1 Components Modelled at the Synchronous and Timing Levels

Three standard components are now described formally at both the synchronous and timing behavioural levels.

4.1.1 Synchronous Level Component Models

The predicates describing the inverter, two-input NOR gate and simple flip-flop at the synchronous level are `INV_S`, `NOR2_S`, `DTYPE12_S`.
These are defined as:

```
INV_S(in,out)         = (∀ t. out t = ¬ in t)

NOR2_S(in1,in2,out)   = (∀ t. out t = ¬ (in1 t ∨ in2 t))

DTYPE12_S(d,q,qbar)   = (∀ t. q(t + 1) = d t) ∧
                        (∀ t. qbar(t + 1) = ¬ q(t + 1))
```

The behaviour of these components is similar to that to that used in previous work. A slight difference in the flip-flop definition is explained in [Herbert86].

4.1.2 Timing Level Component Models

The timing level behaviour of the inverter and two-input NOR gate are described by predicates INV_t and NOR2_t.
These are defined as:

```
INV_t(in,d0,out)        = (∀ t. out(t + d0) = ¬ in t)

NOR2_t(in0,in1,d0,out) = (∀ t. out(t + d0) = ¬ (in0 t ∨ in1 t))
```

We use the master-slave as the timing level flip-flop. The master-slave is described as a structure of propagation delay inverters and NOR gates which fulfills the appropriate internal conditions and whose external parameters can be related to the internal gate delays.

The definition of the predicate MASTER_SLAVE describing the flip-flop is:

```
MASTER_SLAVE(d,clk,q,qbar,setup,hold,mark,space,start,finish) =
  (∃ dbar d0 clkbar d3 a_0 b_0 d1 d2 qa qb d5 d6 a_1 b_1 d4 d7 d8 d9.
    (setup = (MAX(d1,d0 + d2)) + (d5 + d6)) ∧
    (hold = 0) ∧
    (space = setup) ∧
    (mark = d8 + (d9 + ((MAX(d4,d7)) - (MIN(d7,d4))))) ∧
    (start = (MAX(d4,d7)) + (d3 + (d8 + d9))) ∧
    (finish = (MIN(d4,d7)) + d3) ∧
    INV_t(d,d0,dbar) ∧
    INV_t(clk,d3,clkbar) ∧
    NOR2_t(d,clk,d1,a_0) ∧
    NOR2_t(dbar,clk,d2,b_0) ∧
    NOR2_t(a_0,qb,d5,qa) ∧
    NOR2_t(b_0,qa,d6,qb) ∧
    NOR2_t(qa,clkbar,d4,a_1) ∧
    NOR2_t(qb,clkbar,d7,b_1) ∧
    NOR2_t(a_1,qbar,d8,q) ∧
    NOR2_t(b_1,q,d9,qbar) ∧
    MS_INTERNAL_CONDS(d1,d2,d3,d5,d6,d8,d9)
```

The inverter, NOR gate and master-slave flip-flop are modelled exactly as in previous work on timing [Herbert88b].

4.2 Abstraction of NOR Gate

A NOR gate is not a clocked device and so we can choose what clock events define the synchronous behaviour. For the flip-flop example in this section, we

abstract timing level signals by sampling on the rising edges of a clock. We do the abstraction of the NOR gate in the same manner.

A simple case of the abstraction of a NOR gate is when we assume that the input signals are stable for one time unit greater than the propagation delay of the gate.

We have proved the following theorem:

```
STABLE_ABOUT_UP(clk,0)(t0 + del,t1)in0 ∧
STABLE_ABOUT_UP(clk,0)(t0 + del,t1)in1 ⟹

 NOR2_t(in0,in1,del,out) ⟹

  NOR2_S(ABS(UP_OF clk)in0,ABS(UP_OF clk)in1,ABS(UP_OF clk)out) ∧
  STABLE_ABOUT_UP(clk,0)(t0,t1 + del)out
```

This can be paraphrased as follows,

IF

> the inputs are stable from `t0 + del` before rising edges until some time `t1` afterwards, and
>
> the timing level NOR behaviour with delay `del` is satisfied,

THEN

> the input and output signals abstracted on the clock rising edges satisfy the synchronous level NOR behaviour and
>
> the output signal is stable from `t0` before rising edges until `t1 + del` after.

`ABS(UP_OF clk)in0`, `ABS(UP_OF clk)in1` and `ABS(UP_OF clk)out` are synchronous level signals which are derived from the timing level signals `in0`, `in1` and `out` by abstraction on rising edges of the clock `clk`. The synchronous level NOR behaviour (described by predicate `NOR2_S`) holds for these signals.

In Figure 2 we present some waveforms for a timing level NOR gate and the corresponding synchronous level signals obtained by temporal abstraction. Inspection of the synchronous level values shows that the output at a synchronous time step is the *nor* function of the inputs at that time step. This is the purely functional relationship specified for a synchronous level NOR gate by predicate `NOR2_S`.

In Figure 3 we illustrate the part of the theorem of abstraction which relates to signal stability. Signals are specified as being stable over certain intervals of time. Given that the inputs are stable around each clock rising edge, we can deduce that the output is stable over an interval about the same clock edge.

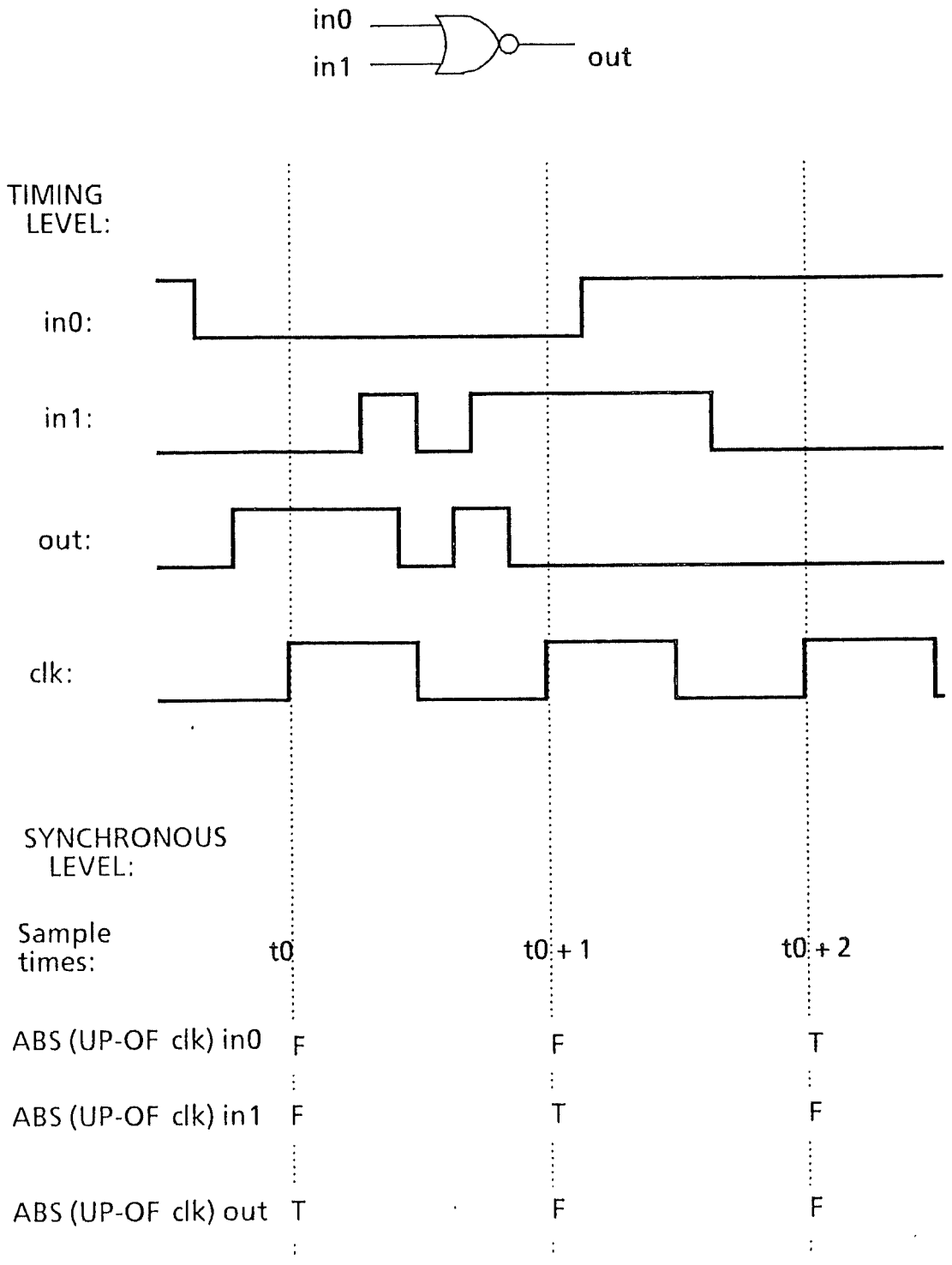

Figure 2: Temporal abstraction of timing level NOR signals

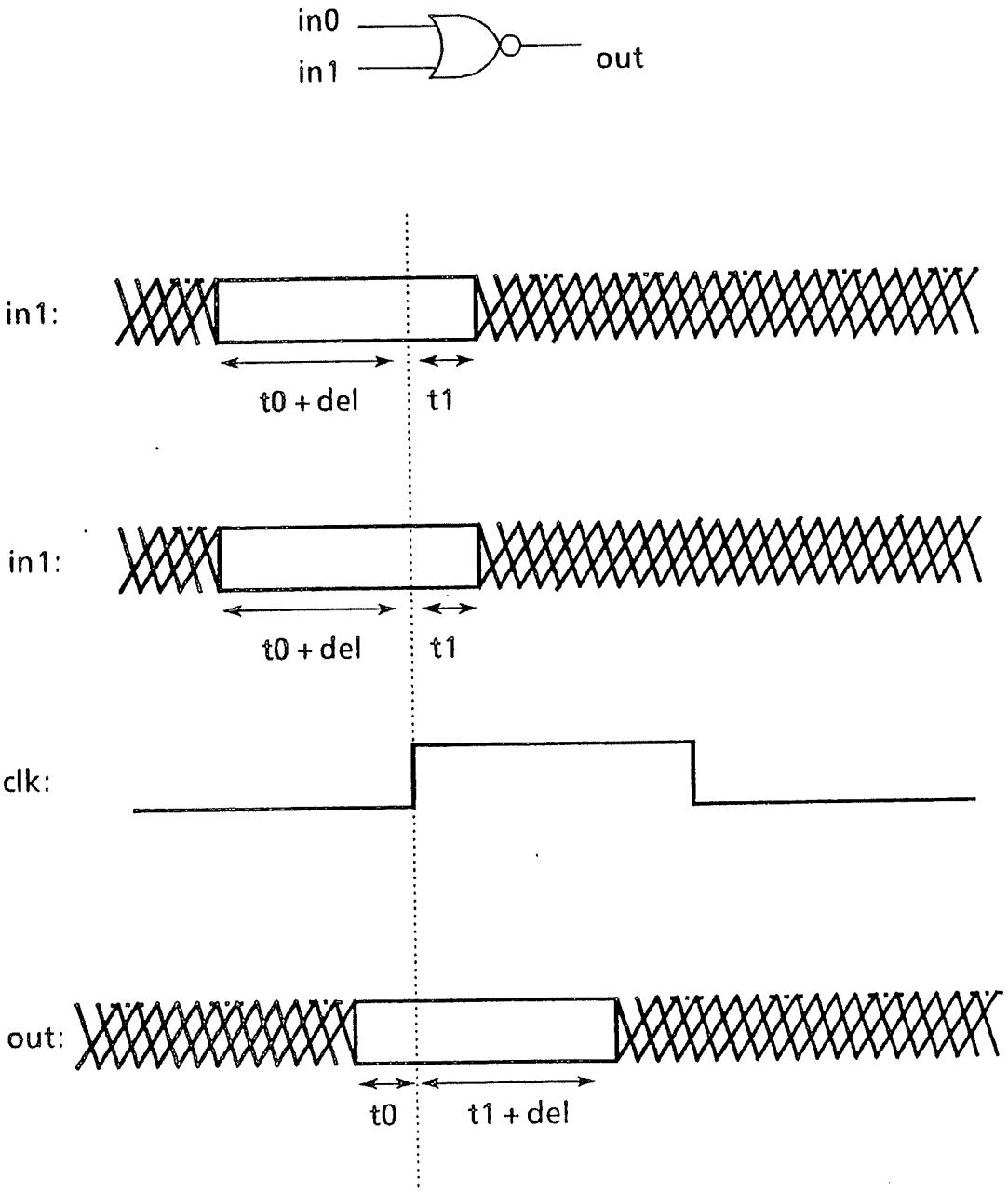

Figure 3: Stability relationship for timing level NOR signals

4.3 Abstraction of Synchronous Flip-flop

The master-slave flip-flop is a clocked device and requires that certain conditions are fulfilled by a well-behaved clock and data signals.
We have deduced the following theorem for the abstraction of the master-slave.

```
STABLE_ABOUT_UP(clk,0)(min_setup,min_hold)d ⟹

 CLOCK_CYCLES(clk,origin,space,mark) ∧
 start < (space + mark) ∧
 min_mark < mark ∧
 min_space < space ⟹

  MASTER_SLAVE
  (d,clk,q,qbar,min_setup,min_hold,min_mark,min_space,start,finish) ⟹

   DTYPE12_S(ABS(UP_OF clk)d,ABS(UP_OF clk)q,ABS(UP_OF clk)qbar) ∧
   STABLE_ABOUT_UP(clk,1)((space + mark) - start,finish)q ∧
   STABLE_ABOUT_UP(clk,1)((space + mark) - start,finish)qbar
```

This can be paraphrased as follows,

IF

the data input is stable from the setup time before rising edges until the hold time afterwards, and

the clk is a regular periodic one with space and mark times which satisfy certain conditions, and

the timing level behaviour of a master-slave implementation holds,

THEN

the abstracted input and output signals satisfy the synchronous level flip-flop behaviour of `DTYPE12_S`, and

the output signals are stable from an interval around all rising clock edges after the first one.

4.3.1 Restrictions on Inputs

The restriction on the data input, `d`, is that it must be stable from `min_setup` before rising edges until `min_hold` after. The clock high and low times, `mark` and `space`, must exceed the minimum high and low times for the flip-flop.

There is also a restriction: `start < (space + mark)`. This condition requires that the period of the clock be greater than the flip-flop timing parameter `start`.

The condition ensures that output changes caused by a clock event occur before the succeeding clock event. This means that we get the correct value on the output when we sample on the clock event times.

This condition was unexpected. A formal mathematical treatment often forces us to re-examine some aspect of a problem. In this case, we had overlooked a slightly obscure but necessary condition. A flip-flop which does not fulfill the restriction can act quite well as a synchronous memory unit but cannot be related by the above abstraction process to unit delay synchronous models.

4.3.2 Stability of Outputs

In the theorem of abstraction for the master-slave, we have deduced certain stability assertions for outputs q and qbar, *i.e.*

```
STABLE_ABOUT_UP(clk,1)((space + mark) - start,finish)q ∧
STABLE_ABOUT_UP(clk,1)((space + mark) - start,finish)qbar
```

The 1 in the above expressions means that q and qbar are stable about all clock rises *after* the first one (cf. definition of STABLE_ABOUT_UP).

To deduce the synchronous behaviour of the flip-flop we assumed that the input signal was stable about all clock rises. *i.e.*

```
STABLE_ABOUT_UP(clk,0)(min_setup,min_hold)d
```

We need to deduce that flip-flop outputs are stable around *all* clock rises if the output of one flip-flop is to provide the input for another flip-flop. To do this we demand that all flip-flop outputs be stable over a certain interval around the first rising clock edge.

The predicate START_STABLE is used to state this condition. This predicate is defined as follows:

```
START_STABLE(t1,t2,clk,q,qbar) =
    STABLE_INTERVAL(t1,t2)(εt. UP_OF clk 0 t)q ∧
    STABLE_INTERVAL(t1,t2)(εt. UP_OF clk 0 t)qbar
```

START_STABLE(t1,t2,clk,q,qbar) states that signals q and qbar are stable from t1 before the first rising edge of clk until t2 after.

We use START_STABLE to define the extra restriction that we are placing on flip-flop implementations. We demand that the flip-flop outputs are stable from time (space + mark) - finish before the first rising edge until start after this edge. This restriction is:

```
START_STABLE((space + mark) - finish,start,clk,q,qbar)
```

By adding this restriction on the flip-flop implementation, we can now deduce that q and qbar are stable around all rising edges *i.e.* :

```
STABLE_ABOUT_UP(clk,0)((space + mark) - start,finish)q ∧
STABLE_ABOUT_UP(clk,0)((space + mark) - start,finish)qbar
```

If all flip-flops fulfill the extra requirement of output stability then all flip-flops satisfy a unit delay synchronous level behaviour such as that defined by DTYPE12_S. This holds even when the flip-flop input signals are generated by another flip-flop.

5 Temporal Abstraction of a Digital Design

Temporal abstraction was used to deduce the synchronous level behaviour of a gate and flip-flop modelled at the timing level. We now consider structures of components modelled at the timing level. An implementation which consists of primitive components modelled at the timing level is called a *timing level implementation.* An implementation consisting of synchronous component models is called a *synchronous level implementation.*

To verify designs modelled at the timing level we must do temporal abstraction of many components. We have devised an efficient procedure to do this. The procedure is well-suited to a life-cycle of functional design and verification followed by the introduction of timing level models. In a later section this procedure is demonstrated by taking an example circuit through all stages of design and verification at the synchronous and timing levels. We firstly describe our perception of the rôle of synchronous and timing level models in the design life-cycle.

5.1 Synchronous and Timing Levels in Design

A design procedure of top-down specification and implementation, followed by bottom-up verification is assumed. However all the ideas discussed apply equally well to other design procedures.

In the initial design of a circuit issues like detailed timing are secondary to achieving the correct functionality. Therefore the initial specification and verification of a circuit can be based on simple synchronous level models. The design can be specified and verified at all levels of the structural hierarchy. A high-level

specification of a device like the ECL chip can be verified using these synchronous level primitive components.

Having verified the function of the circuit at the synchronous level, a particular physical realisation can be chosen and the implementation is mapped onto a structure of physical components. More detailed, timing level models are introduced in order to capture the behaviour of the physical components. We assume that the physical components can be modelled as propagation delay gates. The functional specification is then verified with respect to the timing level implementation. Following this process the high level specification is formally related to an implementation in which the models capture closely the behaviour of physical components.

5.2 Verification of Timing Level Implementation

We must verify that the implementation using components modelled at the timing level achieves the same functional behaviour as the synchronous level implementation. In doing this we must also verify the timing properties of the implementation. Figure 4 depicts the derivation of functional behaviour for a timing level implementation. There are two stages to this procedure.

5.2.1 Temporal abstraction

In the first stage we apply temporal abstraction to modules which consist of primitive components. All these modules must correspond to modules in the synchronous level implementation. The temporal abstraction process separates the functional and timing parts of the behaviour. We deduce that, under certain timing constraints, a functional behaviour holds and some output timing assertions are true.

The function we deduce for a module must satisfy the functional behaviour of the corresponding module in the synchronous level implementation. This condition means that the functional behaviour of all modules in the timing level implementation has been related to the functional behaviour of all modules in the synchronous level implementation. We want to demonstrate that the total timing level implementation achieves the functional behaviour of the synchronous level implementation. Verifying each sub-unit is not sufficient; we must compose the results and verify the top-level of the design.

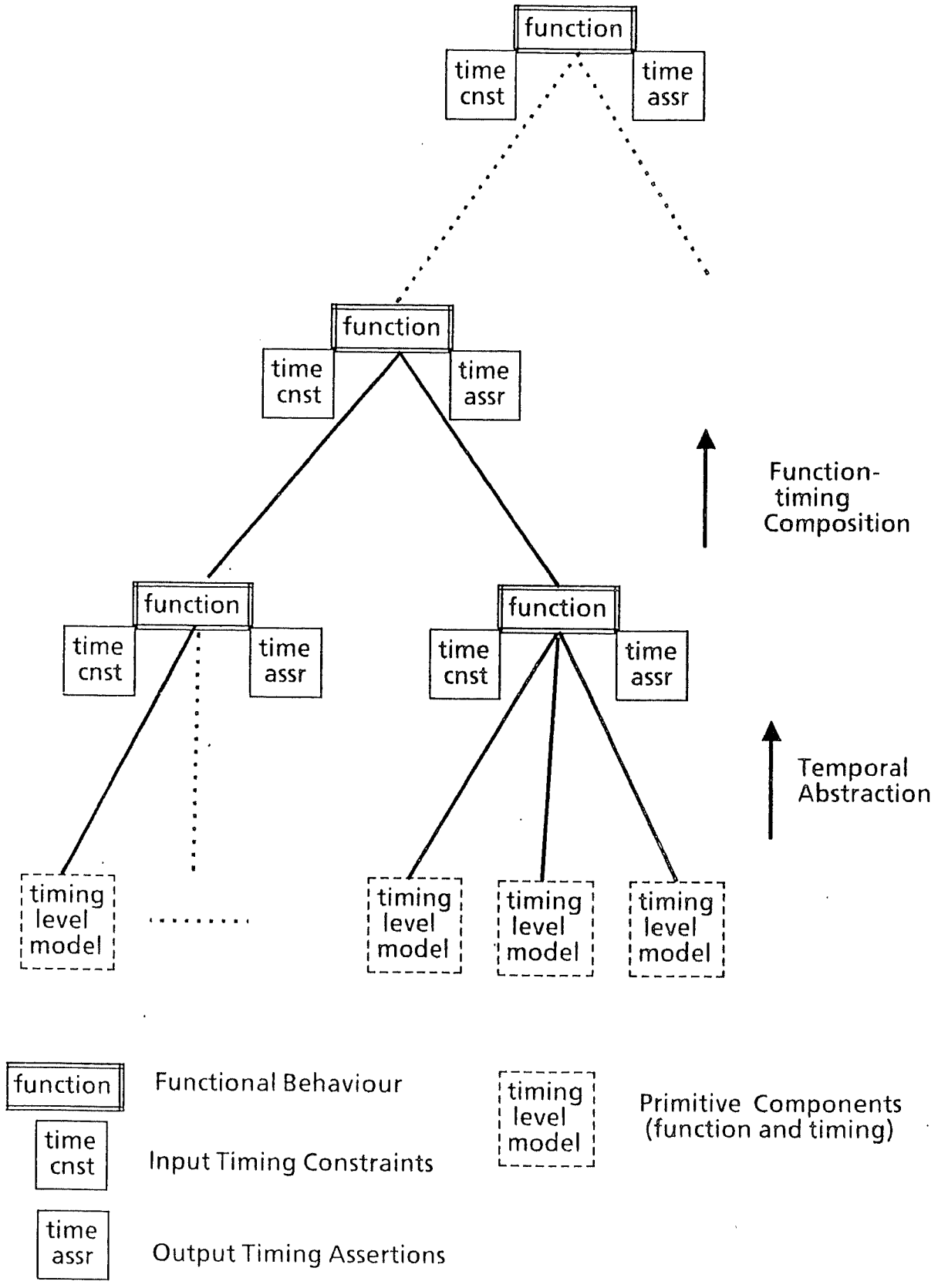

Figure 4: Deriving behaviour of timing level implementation

5.2.2 Function-timing composition

The second stage of verification deals with the function and timing at all higher levels of the design hierarchy. We call the process of deducing the function and timing properties of a device from the function and timing of its components *function-timing composition.* Function-timing composition is used to deduce the behaviour at all higher levels in the structural hierarchy.
Function-timing composition involves:

1. Verifying the timing constraints on interface signals between components.
2. Deducing the timing constraints and timing assertions of the composite device.
3. Deducing the functional behaviour of the composite device.

The main effort in function-timing composition is in verifying timing. The functional behaviour of each module in the timing level implementation has already been related to the functional behaviour of a corresponding part of the synchronous level implementation. The composition of functional behaviour in the synchronous level implementation has already been achieved. By matching functional behaviours to those at the synchronous level we can use the result of composition obtained in the synchronous level proof.

6 Circuit Design Example

We illustrate the use of temporal abstraction in the design life-cycle by taking part of through all stages of design and verification. The circuit is part of the modulator section of the the ECL chip chip ([Hopper86] and is used to detect a modulation error. The circuit is specified, implemented and verified at a synchronous level. A timing level implementation is then related to the previous synchronous implementation by temporal abstraction. We verify that the timing level implementation of the device satisfies the top-level synchronous specification.

The circuit schematic with sub-blocks identified, is given in Figure 5.

The example follows a sequence of:

- Top-down synchronous design, involving specification and implementation.
- Bottom-up verification of synchronous design.

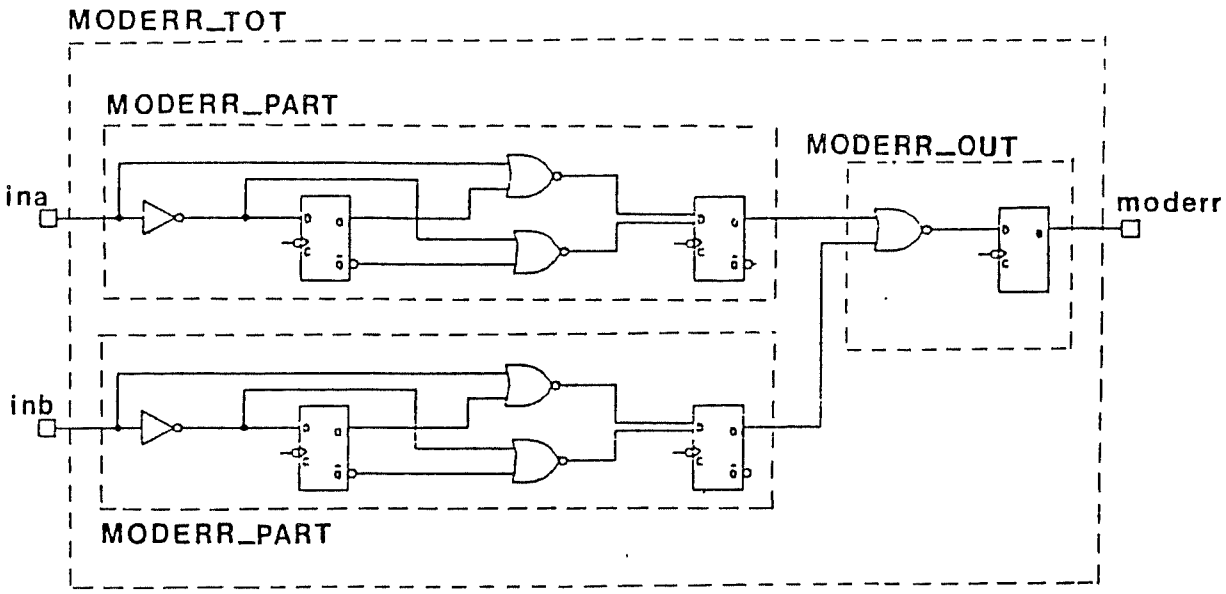

Figure 5: Modulation Error Detector

- Introduction of timing level models of components.
- Temporal abstraction to separate timing properties and function.
- Composition of timing and function throughout the hierarchy.
- Verification of device specification for timing level implementation.

7 Synchronous Level Specification and Verification

The example circuit is specified, implemented and then verified at the synchronous level.

7.1 Top-down Synchronous Design

The circuit is part of an interface chip for the Cambridge Fast Ring [Hopper86]. The modulation system used in the Cambridge Fast Ring is based on delay modulation. In the basic scheme, data can be transmitted using two lines. Boolean values (denoted by `T` and `F`) are encoded by the changes on the lines at successive clock ticks. The value `T` corresponds to changes on both lines; `F` corresponds to a change on one line. Neither line changing is an error, a modulation error. The example circuit detects a modulation error.

The predicates, CHANGED and DEMODULATE_ERR, used in this example are defined as follows:

```
            CHANGED ina t = ¬ (ina t = ina(t - 1))
DEMODULATE_ERR(ina,inb)t = ¬ CHANGED ina t ∧ ¬ CHANGED inb t
```

Top level specification and implementation

We give a specification for the top-level behaviour and then define a structure of sub-blocks to implement this structure.

The specification of behaviour is defined as:

```
MODERR_SPEC(ina,inb,moderr) =
              (∀ t. moderr(t + 3) = DEMODULATE_ERR(ina,inb)(t + 1))
```

The output moderr indicates if a modulation error occurred on inputs ina and inb two time units earlier. The device does not detect errors at time 0, hence the t+1 in the specification.

We choose to implement the circuit as a structure of three sub-blocks. The implementation is defined as:

```
MODERR_TOT(ina,inb,moderr) =
  (∃ qa qb.
    MODERR_PART(ina,qa) ∧
    MODERR_PART(inb,qb) ∧
    MODERR_OUT(qa,qb,moderr))
```

Sub-block specification and implementation

The specifications of behaviour for the sub-blocks are:

```
MODERR_PART_SPEC(ina,q0)       = (∀ t. q0(t + 2) = CHANGED ina(t + 1))

MODERR_OUT_SPEC(qa,qb,moderr) = (∀ t. moderr(t + 1) = ¬ qa t ∧ ¬ qb t)
```

The implementations of these sub-blocks are defined as:

```
MODERR_PART(ina,q0) =
     (∃ 11 12 10 13 14.
       INV_S(ina,10) ∧
       DTYPE12_S(10,11,12) ∧
       NOR2_S(ina,11,13) ∧
       NOR2_S(10,12,14) ∧
       DTYPE21_S(13,14,q0))

MODERR_OUT(qa,qb,moderr) =
     (∃ mod_d.
       NOR2_S(qa,qb,mod_d) ∧
       DTYPE11_S(mod_d,moderr))
```

All the components within the sub-blocks are primitive gates and flip-flops so there is no further decomposition.

We have now constructed a hierarchy with specifications at each level above the lowest primitive component level. We now verify that the structures do achieve the specified behaviours.

7.2 Bottom-up Verification.

We firstly prove that the implementations of the lowest-level blocks satisfy their specifications.
These theorems have been proved:

```
MODERR_PART(ina,q0) ⟹ MODERR_PART_SPEC(ina,q0)

MODERR_OUT(qa,qb,moderr) ⟹ MODERR_OUT_SPEC(qa,qb,moderr)
```

The implementations consisting of synchronous level gates and flip-flops have been proved correct with respect to the specifications of the blocks. We can now use the specification for each block rather than the implementation when verifying behaviour at the next level in the hierarchy.

The next level is the top level. We have proved that the top-level specification is satisfied by the implementation:

```
MODERR_TOT(ina,inb,moderr) ⟹ MODERR_SPEC(ina,inb,moderr)
```

This theorem states that the implementation, consisting of synchronous elements at the lowest level, satisfies the specification of behaviour defined by predicate MODERR_SPEC. The specification and verification of the device at the synchronous level is now complete.

8 Timing Level Implementation

We now implement the example device using components modelled at the timing level rather than at the synchronous level. We use actual values derived from a gate-array data sheet (Appendix 1) for the propagation delays of gates. The timing parameters for the master-slave flip-flops are derived from the delays of its component gates. These delays are got from the same data sheet. Thus, the timing level implementation corresponds directly to a structure of propagation delay gates such as might comprise a gate array realisation.

8.1 Implementation Structure

The structural hierarchy of the timing level implementation is exactly the same as the synchronous level implementation. The primitive components are now modelled differently.

We define the predicates MODERR_PART_t and MODERR_OUT_t to describe the structures at the timing level which correspond to the synchronous level implementations MODERR_PART and MODERR_OUT.
These are defined as follows:

```
MODERR_PART_t(ina_t,clk,space,mark,qa_t) =
  (∃ 10_t 11_t 12_t 13_t 14_t qa_0b_t.
    INV_t(ina_t,73,10_t) ∧
    MASTER_SLAVE(10_t,clk,11_t,12_t,257,0,156,257,257,101) ∧
    NOR2_t(ina_t,11_t,53,13_t) ∧
    NOR2_t(10_t,12_t,53,14_t) ∧
    MASTER_SLAVE_2
    (13_t,14_t,clk,qa_t,qa_0b_t,257,0,156,257,257,101) ∧
    START_STABLE((space + mark) - 257,101,clk,11_t,12_t) ∧
    START_STABLE((space + mark) - 257,101,clk,qa_t,qa_0b_t))
```

and

```
MODERR_OUT_t(qa_t,qb_t,clk,space,mark,moderr_t) =
  (∃ mod_d_t modbar_t.
    NOR2_t(qa_t,qb_t,53,mod_d_t) ∧
    MASTER_SLAVE
    (mod_d_t,clk,moderr_t,modbar_t,257,0,156,257,257,101) ∧
    START_STABLE((space + mark) - 257,101,clk,moderr_t,modbar_t))
```

All the timing level components are in a 1-to-1 correspondence with similar components in the synchronous level implementations. This is not necessary; we just require that the timing level components achieve the same functional behaviour as the synchronous level implementation.

The non-component terms in the timing level implementation, involving the predicate START_STABLE, state the requirements that flip-flop outputs must be stable before the first clocking edge. The use of these extra terms in the timing level implementation has been explained in section 4.3.2.

9 Verification of Timing Level Implementation

We now verify that the implementation using components modelled at the timing level achieves the same functional behaviour as the synchronous level implementation. In doing this we also verify the timing properties of the implementation. The

derivation of functional behaviour for a timing level implementation is depicted in Figure 4. There are two stages to this procedure: temporal abstraction and function-timing composition.

9.1 Temporal Abstraction

The first stage of the verification process involves performing temporal abstraction of structures of components modelled at the timing level.
These theorems follow the pattern of:

```
input stability conditions  ⟹

        clocking conditions      ⟹

            timing level behaviour ⟹

              synchronous level behaviour  ∧
              output stability assertions
```

We have proved theorems of abstraction for the two timing level structures MODERR_PART_t and MODERR_OUT_t.

The theorem of abstraction for MODERR_PART_t is:

```
STABLE_ABOUT_UP(clk,0)(ina_t_t0,ina_t_t1)ina_t ∧
383 ≤ ina_t_t0  ⟹

  CLOCK_CYCLES(clk,origin,space,mark) ∧
  156 < mark ∧
  257 < space ∧
  567 ≤ (space + mark) ⟹

    MODERR_PART_t(ina_t,clk,space,mark,qa_t) ⟹

      MODERR_PART(ABS(UP_OF clk)ina_t,ABS(UP_OF clk)qa_t) ∧
      STABLE_ABOUT_UP(clk,0)((space + mark) - 257,101)qa_t
```

The theorem of abstraction for MODERR_OUT_t is:

```
STABLE_ABOUT_UP(clk,0)(qa0_t_t0,qa0_t_t1)qa_t ∧
STABLE_ABOUT_UP(clk,0)(qb0_t_t0,qb0_t_t1)qb_t ∧
310 ≤ qa0_t_t0 ∧
310 ≤ qb0_t_t0 ⟹

CLOCK_CYCLES(clk,origin,space,mark) ∧
156 < mark ∧
257 < space ⟹

MODERR_OUT_t(qa_t,qb_t,clk,space,mark,moderr_t) ⟹
```

```
MODERR_OUT
 (ABS(UP_OF clk)qa_t,ABS(UP_OF clk)qb_t,ABS(UP_OF clk)moderr_t) ∧
STABLE_ABOUT_UP(clk,0)((space + mark) - 257,101)moderr_t
```

We describe the theorem for `MODERR_OUT_t`.

Input signals `qa_t` and `qb_t` must be stable from $31ns$ before rising edges until some (unconstrained) time afterwards. The clock signal must be a regular one with high and low times greater than $15.6ns$ and $25.7ns$ respectively. These restrictions on clock high and low times can be translated into a minimum clock period of $41.3ns$ and a mark-space ratio of approximately 3:5.

Given that the input and clocking constraints are satisfied, we can deduce that the timing level implementation, `MODERR_OUT_t`, satisfies the synchronous level behaviour, `MODERR_OUT` for signals `qa_t`,`qb_t` and `moderr_t` abstracted on rising clock edges.

We also deduce that the output `moderr_t` is stable around rising edges of the clock for an interval determined by the clock period `mark + space`. If `mark` and `space` fulfill the minimum clocking requirements, then `moderr_t` is stable from $15.6ns$ before edges until $10.1ns$ afterwards.

The temporal abstraction process has enabled us to separate timing and function. We deduce for groups of components modelled at the timing level:

- Timing properties.
  - Timing constraints consisting of input stability conditions and clocking conditions.
  - Timing assertions stating when outputs are stable.
- Function *i.e.* the behaviour at the synchronous level.

The temporal abstraction of modules is tedious but not difficult. We have automated the process for the combinational parts of the design. If the timing level models of components do not match the synchronous ones than some manipulation will be necessary to prove that the timing level achieves the synchronous level behaviour. We do not require equivalence, just that the timing level behaviour implies the synchronous level.

9.2 Function-timing Composition

In the first stage of timing level verification the function and timing properties of modules have been deduced. Function-timing composition is now used to verify

all higher levels of the design hierarchy. Function-timing composition derives the function and timing of a part of the design from the function and timing of its components.

In function-timing composition the timing constraints on interface signals are also verified. A theorem stating the function and timing of a module will usually have stability constraints on inputs and will assert when outputs are stable. In function-timing composition, the assertions of output stability for one module can be used to verify the input stability constraints of another. If one fails to verify these timing requirements, then redesign or adjustment of component delays may be necessary.

The timing level implementation at the next level is described by predicate MODERR_TOT_t. This is defined by:

```
MODERR_TOT_t(ina_t,inb_t,clk,space,mark,moderr_t) =
  (∃ qa_t qb_t.
    MODERR_PART_t(ina_t,clk,space,mark,qa_t) ∧
    MODERR_PART_t(inb_t,clk,space,mark,qb_t) ∧
    MODERR_OUT_t(qa_t,qb_t,clk,space,mark,moderr_t))
```

The timing level implementation is a structure of sub-blocks, each of which corresponds to a similar sub-block in the synchronous design. Above the level of primitive components, there is a direct correspondence between all modules in the timing and synchronous level implementations.

Deducing the function and timing properties of MODERR_TOT_t involves composition of the function and timing properties of its sub-blocks MODERR_PART_t and MODERR_OUT_t. The conditions of stability for the inputs of MODERR_OUT_t are verified by the stability assertions on the outputs of both MODERR_PART_t modules. Therefore in the theorem stating the behaviour of the composite device we just have constraints on its external inputs.
The following theorem stating function and timing properties of MODERR_TOT_t has been proved.

```
STABLE_ABOUT_UP(clk,0)(inb_t_t0,inb_t_t1)inb_t ∧
383 ≤ inb_t_t0 ∧
STABLE_ABOUT_UP(clk,0)(ina_t_t0,ina_t_t1)ina_t ∧
383 ≤ ina_t_t0 ⟹

  CLOCK_CYCLES(clk,origin,space,mark) ∧
  156 < mark ∧
  257 < space ∧
  567 ≤ (space + mark) ⟹

   MODERR_TOT_t(ina_t,inb_t,clk,space,mark,moderr_t) ⟹
```

```
MODERR_TOT
(ABS(UP_OF clk)ina_t,ABS(UP_OF clk)inb_t,ABS(UP_OF clk)moderr_t) ∧
STABLE_ABOUT_UP(clk,0)((space + mark) - 257,101)moderr_t
```

The predicate MODERR_TOT describes the behaviour of the synchronous level implementation of the example device. We have therefore proved that the timing level implementation achieves the same functional behaviour as the synchronous level implementation.

9.3 Statement of Correctness

We have already proved that the synchronous level implementation satisfies the device specification:

```
MODERR_TOT(ina,inb,moderr) ⟹ MODERR_SPEC(ina,inb,moderr)
```

We have now deduced that the synchronous level behaviour holds for the abstracted signals of the timing level implementation. We can therefore deduce, using a single rule of inference, that the device specification holds for the abstracted signals of the timing level implementation. This theorem is presented in Figure 6.

```
STABLE_ABOUT_UP(clk,0)(inb_t_t0,inb_t_t1)inb_t ∧
383 ≤ inb_t_t0 ∧
STABLE_ABOUT_UP(clk,0)(ina_t_t0,ina_t_t1)ina_t ∧
383 ≤ ina_t_t0 ⟹

  CLOCK_CYCLES(clk,origin,space,mark) ∧
  156 < mark ∧
  257 < space ∧
  567 ≤ (space + mark) ⟹

   MODERR_TOT_t(ina_t,inb_t,clk,space,mark,moderr_t) ⟹

     MODERR_SPEC
     (ABS(UP_OF clk)ina_t,ABS(UP_OF clk)inb_t,ABS(UP_OF clk)moderr_t)
```

Figure 6: Correctness of timing level implementation

The inputs ina_t and inb_t must be stable from $38.3ns$ before rising edges of the clock until some time afterwards. The clock must have a period greater than $56.7ns$ and the minimum lengths it must remain high and low are $15.6ns$ and $25.7ns$ respectively. If these constraints are satisfied then the timing level

implementation ensures that signals `ina_t`, `inb_t` and `moderr_t` abstracted on the rising edges of the clock signal `clk` satisfy the functional specification of the device.

This ends the verification of the example device. We have proved that an implementation, using component models which capture closely the physical device behaviour, satisfies a high-level functional specification. We have also deduced the timing constraints on the input signals to the device and the minimum clock high and low times.

Modelling closely the physical behaviour has a number of advantages. We can feel more confident that the physical device will achieve the functional behaviour. We know the timing constraints on the input signals and the maximum clock frequency.

10 Discussion

10.1 Relationship to Timing Analysis

The process of generating and verifying the timing conditions of a circuit is called *timing analysis*. Timing analysis is important at LSI and VLSI levels of integration. The physical device has a scarcity of probe points; timing problems can be inadvertently introduced when sub-blocks of a large circuit are combined; the layout contributes to timing problems.

In performing temporal abstraction we generate a number of timing conditions which must hold if the timing level circuit is to achieve the synchronous level behaviour. Given the actual signal timing parameters and circuit delays we can deduce whether these conditions hold. Formally deducing and verifying these timing conditions is a similar process to non-formal timing analysis.

Important activities in timing analysis are:

1. Determining whether paths in the design meet stated timing criteria.
   For example, data signals must arrive at clocked elements in time for valid sampling but not too early (*i.e.* set-up and hold times must be observed).

2. Calculating minimum clock period.

3. Identifying critical paths (*i.e.* paths whose delay is a limiting factor to the minimum operating speed of the circuit).

The first two activities form part of formal timing verification. Constraints on the stability of input signals are determined by timing requirements such as set-

up and hold times, and the path delays. Deducing and verifying these timing constraints corresponds to the first activity.

The second activity is part of formal timing verification. We deduce the minimum clock high and low times, and the minimum clock period. For example, the modulation error device has a minimum clock period of $56.7ns$ and minimum clock high and low times of $15.6ns$ and $25.7ns$ respectively. (Some devices may only have clock high and low constraints; the minimum clock period is therefore the sum of the minimum clock high and low times.)

Identifying critical paths has not formed part of temporal abstraction. However the derived timing constraints could be analysed for this purpose.

10.2 Advantages of Formal Timing Verification

Formal verification of timing conditions allows us to deduce a more abstract, synchronous level behaviour for a structure of components modelled at the timing level. The synchronous level behaviour provides the basis for the more usual functional verification of the implementation. We therefore explicitly relate the process of timing verification to functional verification. Although we separate timing and functional behaviour, both are based on unique timing level models of components. Non-formal timing analysis is an independent part of the digital design process and is not formally related to functional verification.

The formal verification is compositional and exploits the design hierarchy. The timing properties of a device can be deduced from the timing properties of its components. When modules are composed timing constraints on interface signals must be verified or added to the constraints of the composite device. Enforcing these constraints means that only valid compositions are achieved. Verification of timing properties hierarchically indicates the source of timing errors and bottlenecks. For example, the minimum clock period of modules `MODERR_PART_t` and `MODERR_OUT_t` are $56.7ns$ and $41.3ns$ respectively. Thus, `MODERR_PART_t` limits the speed of the composite device `MODERR_TOT_t`.

The formal verification of timing can be extended to include function. When we separate timing and function we use the worst case timing behaviour, ignoring function. The basic models of components describe both the timing and functional behaviour and so we can include functional information without changing these models. Taking function into account, the timing behaviour of a NOR gate might state that a stable *high* on any input guarantees output stability. Different parts of the design can be verified with or without functional information in a consistent

fashion due to the unique underlying models of components. The use of functional information is of practical importance because bogus timing problems are found when function is ignored.

10.3 Summary

We have devised a procedure of temporal abstraction followed by function-timing composition to derive the behaviour of the timing level implementation. Efficiency is achieved by separating timing and function, and relating the functional behaviour of the timing level implementation to that of the synchronous level design. The procedure is modular and hierarchical and therefore the sources of timing errors and bottlenecks are clearly indicated.

The formal verification of timing fits into a design life-cycle of functional design and verification, followed by timing level implementation and verification. Proofs based on synchronous level models can be done independently of the timing level models. When the timing level models of components are introduced, it is not necessary to repeat *any* proofs of correctness of the functional behaviour.

Verifying the functional behaviour of a timing level implementation involves generating and verifying various timing properties. We have identified this process with non-formal timing analysis of circuits. In contrast to non-formal timing analysis, the formal verification of timing conditions is explicitly related to the verification of functional behaviour. As far as we know, the idea that timing analysis provides a link between detailed timing models and synchronous behavioural models has not been noted or formalised previously.

References

[Cohn87] A. Cohn, "A Proof of Correctness of the Viper Microprocessor: The First Level", Technical Report No. 104, Computer Laboratory, University of Cambridge, U.K., 1987.

[Gordon85a] M. J. C. Gordon, "HOL A Machine Oriented Formulation of Higher Order Logic", Technical Report No. 68, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1985.

[Gordon85b] M. J. C. Gordon, "Why Higher-Order Logic is a Good Formulism for Specifying and Verifying Hardware", Technical Report No.

77, Computer Laboratory, University of Cambridge, Cambridge, U.K., 1985.

[Hanna85] F. K. Hanna and N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study", Internal Report, University of Kent, U.K., 1985.

[Herbert86] J. M. J. Herbert, "Application of Formal Methods to Digital System Design", *Ph D Thesis*, University of Cambridge, U.K., 1986.

[Herbert88a] J. M. J. Herbert, "Cambridge Fast Ring ECL Chip Case Study using HOL", Technical Report, Computer Laboratory, University of Cambridge, U.K., 1988.

[Herbert88b] J. M. J. Herbert, "Verification of Basic Memory Devices", Technical Report, Computer Laboratory, University of Cambridge, U.K., 1988.

[Hopper86] A. Hopper and R. M. Needham, "The Cambridge Fast Ring Networking System", Technical Report No. 90, Computer Laboratory, University of Cambridge, U.K., 1986.

[Melham87] T. Melham, "Abstraction Mechanisms for Hardware Verification", *Specification, Verification and Synthesis, Proceedings of the Workshop on Hardware Verification*, Calgary, Canada, edited by: G. Birtwistle and P. Subrahmanyam, 1987.

[Hunt85] W. A. Hunt Jr., "FM8501: A Verified Microprocessor", Technical Report 47, University of Texas at Austin, December 1985.

Appendix 1

Delay of Gates in Bipolar Gate Array

Typical gate delay in nanoseconds as a function of fan-in and fan-out.

Fan-out

Fan-in	1	2	3	4	5	6
1	4.8	7.3	9.8	12.3	14.9	17.4
2	5.3	7.8	10.3	12.9	15.4	17.9
3	8.1	10.7	13.2	15.7	18.2	20.7
4	8.7	11.2	13.7	16.2	18.7	21.3
5	11.5	14.0	16.5	19.1	21.6	24.1
6	12.0	14.5	17.1	19.6	22.1	24.6
7	14.9	17.4	19.9	22.4	24.9	27.5
8	15.4	17.9	20.4	22.9	25.5	28.0