

# Formal Synthesis\*

Michael P. Fourman<sup>†</sup>

M.Fourman@ed.ac.uk

&

Roberto A. Hexsel<sup>‡</sup>

rh@lfc.ed.ac.uk

Laboratory for the Foundations of Computer Science  
Edinburgh University.

March 1991

## Abstract

Most applications of formal proof to digital hardware have been aimed at proving the correctness of existing designs. Here, we address the application of formal methods to the *design* of digital systems. Our particular concern is to show how a formal approach can address the problems of system level design: complexity and abstraction. To relate the system-level behaviour to the behaviours of primitive components, we model and relate behaviours at different levels of abstraction.

We base our presentation on a small, but revealing, case-study; starting from a high-level specification, we design a digital stopclock. The stopclock is specified in terms of user-level time ticks, and is implemented in terms of a much faster system-level clock. A block diagram of the design is formalised at the abstract level. Abstractions are introduced to relate this to the low-level implementation. The ‘glue’ logic, necessary to connect components using different abstractions, is formally derived.

---

\*This is a revised version of the article in the proceedings of the Fourth Higher Order Workshop, Banff, 1990; G. Birtwistle (ed.), Springer Verlag, 1991.

<sup>†</sup>This work was partially supported by SERC grant GRF35890.

<sup>‡</sup>Supported by a grant from CAPES, Ministry of Education, Brazil.

# 1 Introduction

The recent VLSI revolution is based on the MOS transistor, which may be modelled as a switch. Using this as a primitive, we build gates, registers, counters, state machines, controllers, microprocessors and systems. To understand the behaviour of a system as a function of the behaviours of its component parts, we need to model and relate behaviours at many levels. Models based on predicate logic and formal proof have been widely used to verify the correctness of digital circuits. A few examples are [Gordon85] [CGM86] [Gordon87] [Cohn87] [Hunt87] [Cohn89] [HLD89] [BG90]. In most of these works, the behaviour of a device is modelled by a predicate that expresses the constraints the device imposes on the signals on its external ports.

In this paper, we address the application of formal methods to the *design* of digital systems, continuing a programme started in [FPZ88], and developed in [FM89] [FFFH90] [Fourman90]. Our particular concern is to show how a formal approach may address the problems of system level design: complexity and abstraction. We base our presentation on a small, but revealing, case-study. Starting from a high-level specification, we design a digital stopclock. Consider the following informal specification:

*Design a digital stopwatch with a three digit, seven-segment display (to read seconds, tenths of seconds, and tens of seconds), and two control buttons, ‘reset’ and ‘start/stop’. When the reset button is pressed the display is cleared. The start/stop button is used to start and stop the clock.*

*You are provided with a 1MHz system clock, and a 10Hz digital signal synchronised with the system clock. You may assume that the buttons also produce synchronised digital signals.*

We will:

- formalise the specification at the user level (10Hz clock);
- write abstraction functions to express the controls and display in terms of 1MHz signals;
- develop an implementation by refinement.

The interest of the exercise is to see that design can be started at a high level of abstraction. We then introduce the abstractions used to relate the high-level specification to a concrete implementation. These relate the two levels of time. The control signals are *asynchronous* with respect to the specification-level model of time, and the output must satisfy constraints imposed by other system components; in this case the output is the display, and the behaviour of the human eye requires a large ‘hold time’, if the display is to be perceived.

## 1.1 Modelling Behaviour

We begin with a review of established methods for the application of formal logic to the specification of digital behaviour, and then continue with our design.

To model the behaviour of combinational logic, we model an output signal as a boolean function of the signals on the inputs. Function composition models the connection of the output of one device to an input of another. Tools, based principally

on Boolean algebra, allow us to implement a given function quickly, accurately and efficiently. A similar, functional model may be used at a higher level of abstraction, to reason about simple data-flow architectures. To model general patterns of connection, and the hiding of internal wires, we need more-sophisticated models of behaviour. For instance, we may not wish to distinguish inputs from outputs. The classic example is a MOS transistor, sometimes used as a bidirectional switch. Informally, we model its behaviour as a switch controlled by the signal on the gate. Although the gate can reasonably be regarded as an input, both source and drain can act as inputs or outputs. We cannot formalise this model as a function.

At higher levels of abstraction, we may want to allow the possibility that, for some inputs, the output is indeterminate. To represent a unit delay as a function, we have to say what logic value appears on the output at time 0. It is more realistic to say simply that all we know of the behaviour is that the value on the input at time  $t$  appears on the output at time  $t + 1$ . This is not a functional relationship; the signal on the output is not determined by the signal on the input — it is however constrained by it.

We model behaviour by saying that a device only allows certain combinations of signals on its ports. This representation of the behaviour as a subset of the possible combinations of signals is analogous to the representation of the analogue behaviour of a power source by a curve showing what combinations of current and voltage it admits. The characteristics of a load are represented similarly, by another curve. To determine what happens if we connect the two, we intersect the two curves. This model works well — but it is only a model and should be applied with care; it will not tell us what happens when we connect a toy 12V DC motor directly to the mains voltage.

We model the interconnection of digital devices in the same way: the behaviour of a component is represented as a subset of the possible combinations of signals at its ports; when we compose components, their combined behaviour is given by intersecting their individual behaviours. Again, this is only a model, and it must be applied with care; it will not correctly predict the behaviour of a circuit that wantonly violates design rules.

Clearly, graphical or tabular methods do not provide a practical way of manipulating these subsets. However, formal logic provides a notation for describing and manipulating these models. Predicate Logic provides a formal language for expressing relationships between things, together with a mathematical model of deductive reasoning about these relationships. The behaviour of a device may be described by a formula in predicate logic; the behaviour of a complex module may be composed from the behaviours of its parts, using the logical operations, *conjunction*,  $\wedge$ , and *existential quantification*,  $\exists$ , to model connection and hiding of internal wires. We use notation borrowed from ML [MTH90] to specify types and functions, and leave it to the reader to translate this to her favourite formalisation of typed predicate calculus.

To analyse continuous change, it is usual to model time using real numbers. When modelling digital behaviour, we may only be interested in the discrete sequence of states of a system; or we may require more detailed models of time that allow us, for example, to talk about setup and hold times, or concurrent systems. In this paper, we take a simple view of time as a potentially infinite sequence of clock ticks, modelled by the natural numbers —  $t \in \mathbf{nat}$  represents the number of abstract *clock ticks* since an (implicit) global reset. The time-dependent sequence of values on a wire is represented as a function from times to values.

```
type time = nat ;
type 'a signal = time → 'a ;
```

Representing behaviours by relating *signals* on the ports of a device makes the representation of state implicit.

This model of behaviour and composition is only valid under appropriate constraints on the circuits we consider, and on the the interpretation of our model. Design rules express some such constraints, others must be checked by lower-level analysis of details of timing and electrical behaviour. We use formal methods to reason about the functional behaviour of devices operating in environments that allow us to use a synchronous digital abstraction of behaviour. Other tools are needed to verify these environmental constraints.

## 1.2 Specification

Specifications are essential contractual and methodological tools. The primary use of a behavioural model is to *specify* the required behaviour. This specification is central to the work of an engineer; his basic task is to design a system that meets the specification. Because the specification provides the interface between customer and designer, it must be understood by both. Specifications have another, equally important, rôle. Specifications express abstraction; we may use a component without worrying about (or even knowing about) its inner workings, if we know that it satisfies its specification. This *behavioural abstraction* is crucial to the design of complex systems; it provides the basis for managing complexity using hierarchical decomposition. Specifications are also necessary for scientific project management; once the specification of a component has been fixed, its users and implementers can work in parallel. Well-specified components also form the basis for reusable libraries.

Product specification and design documentation are often confused. A *specification* specifies a behaviour which may be implemented in different ways. Any user relying on the specification should be able to substitute one implementation for another without affecting the correctness of the overall system. Design *documentation* describes an implementation, it is necessary for design debugging, maintenance and test-pattern generation, and documents features incidental to that particular implementation. Using design documentation as a basis for incorporating a component in a system is usually a mistake — if you rely on an incidental feature you have no freedom to improve your system design by later substituting a different implementation of the component specification, unless, by chance, it shares the ‘feature’. Worse, the original part may become unavailable, being superseded by another that meets the specification but has different behaviour, and you may have to redesign your system.

## 1.3 Correctness and Abstraction

Current “silicon compilers” take as input a register-transfer level description of behaviour, in terms of concrete data representations. There is usually no machine-assistance for the refinement of a design to this, register-transfer, level. Without proper documentation of the abstractions used, design management is difficult. Moreover, bugs introduced during the refinement process can be subtle, and dangerous.

Specifications may express abstraction, by hiding implementation detail. Another type of abstraction is expressed by *abstraction functions*. The need for these arises as soon as our specifications are expressed at a level higher than the implementation. The specification stipulates a relation that must hold between abstract objects. In an implementation, these abstract objects will be represented by patterns of concrete

data in space and time. We use *interface specifications* to express the *data abstraction* and *temporal abstraction* which relate the concrete signals of an implementation to the abstractions of the design specification. To express the abstractions that relate signals at differing levels of description we again use predicate logic.

In developing our case study, we will look in some detail at various examples of abstraction. Here, we make one simple, but subtle, point. A given concrete signal usually represents at most one abstract signal, so it is possible to think of abstraction as a function, from concrete to abstract signals. However, not all concrete signals will correspond to one of our *intended* abstract values. We have to take account of the fact that our abstract signals (sequences of abstract values, modelled as functions on time) may at some times take values outwith the datatype used in the abstract specification.

One solution would be to reason explicitly about partial functions and terms that may not denote; some patterns of concrete data may not correspond to *any* abstract value. Our current approach is slightly different; an implementation of a datatype from the specification is given by *embedding* the type of specification-level signals in the type of concrete signals — some patterns of concrete data may not correspond to any *proper* abstract value. While reasoning about the abstract level, we take account of the possibility of ‘illegal’ values.  $\mathbf{E} \tau$  (read  $\tau$  *exists*) expresses the assertion that  $\tau$  denotes a value within the intended range — this assertion may be false. Quantifiers are (implicitly) restricted to this range, and we interpret equality to imply existence:

$$\begin{aligned} (\forall x \cdot \phi) &\hat{=} \forall x \cdot \mathbf{E} x \Rightarrow \phi & (\exists x \cdot \phi) &\hat{=} \exists x \cdot \mathbf{E} x \wedge \phi \\ (\tau = \sigma) &\hat{=} \mathbf{E} \tau \wedge \mathbf{E} \sigma \wedge \tau = \sigma \end{aligned}$$

We adopt the same convention for atomic relations; for example, if a datatype has an order then we assume this is extended vacuously to illegal values:

$$(\tau < \sigma) \hat{=} \mathbf{E} \tau \wedge \mathbf{E} \sigma \wedge \tau < \sigma$$

## 2 Formally Specifying the Stopclock

We now formalise user-level specification. The user’s view of the stopclock is that it changes state on each 1/10th sec. clock tick. We model user-level clock ticks as naturals, the display, the reset command and the start/stop command are modelled as signals `display`, `reset` and `stst` that depend on this implicit 10Hz clock.

```
type Utime = nat ;
type 'a USignal = Utime → 'a ;
```

`reset` and `stst` are boolean signals; for the `display`, the user sees (or hopes to see) a sequence of three-digit values:

```
datatype Digit = 0 | 1 | ... | 8 | 9 ;
type Display = {tens:Digit, secs:Digit, tenths:Digit} ;
display: Display USignal ;
```

The specification relates `display`, `reset` and `stst`. For instance, we may specify that, at each clock tick, if the clock is running, unless the clock is reset the display is incremented; if the stopclock is not running, then the display is unchanged unless the clock

is reset. To formalise the implicit state (is the clock running or not), we introduce an internal boolean signal, `run`. We specify how the next state and output depend on the present state and inputs, and how the stopclock looks when we switch it on:

```
val AbsS(reset, stst, display) =
  ∃ run:bool USignal. ∀ τ:Utime.
    display(0) = {tens=0,secs=0,tenths=0}
    ∧ display(τ+1) = if (reset τ) then {tens=0,secs=0,tenths=0}
                      else if (run τ) then nextTime(display τ)
                      else (display τ)
    ∧ run(0) = false
    ∧ run(τ+1) = if (reset τ) then false
                  else if (stst τ) then not(run τ)
                  else (run τ) ;
```

`nextTime` is defined by:

```
fun nextTime{tens,secs,tenths} =
  if (tenths < 9) then {tens,secs,tenths=nextDigit tenths}
  else if (secs < 9) then {tens,secs=nextDigit secs,tenths=0}
  else if (tens < 5) then {tens=nextDigit tens,secs=0,tenths=0}
  else {tens=0,secs=0,tenths=0} ;
```

Here, `nextDigit` is a partial function defined in the obvious way for  $n < 9$ , and `Digits` have the usual ordering. There is a dependency between the values of `run`, `stst` and `display`: `stst τ` influences `run(τ+1)` which in turn influences `display(τ+2)`. Notice that the user-level specification is deterministic since it defines the behaviour of the stopclock for all combinations of the control signals.

### 3 Implementing the Stopclock

We are also given some implementation dependent information; the implementation will be in terms of 1MHz signals. The top-level specification, quite properly, ignores the 1MHz clock: the user of a stopwatch shouldn't have to be aware of its existence. To design an implementation, we will have to decide how to represent the user-level abstract signals concretely. For example, the high-level specification does not explain that the reset button may be pressed momentarily, asynchronously with respect to the 10Hz clock tick, nor that the output must remain well defined for long enough to allow the human eye, which forms part of the total system, to perceive the correct display. To refine the specification, we will formalise the relation of the abstract sequence of displayed values, to the the low-level sequence that can, potentially, change too quickly to be read. We also address the internal representation of reset and start/stop — relating the abstract signals of the specification to concrete signals on wires attached to the buttons.

**Classifying Abstractions** There are infinitely many different abstraction functions we could use. However, many are common enough to be considered clichés, and Melham's work, [Melham88], shows that some systematic study is profitable. In general, we must explain how the high-level data of the specification is represented in terms of lower levels. It is useful, but often impossible, to factor this into separate timing and

data abstractions. Melham presents four kinds of abstraction: *structural*-, *behavioural*-, *data*- and *temporal*-. These are all used in our development of the case study; we also have to introduce some *mixed* abstractions. We believe that these occur frequently in real designs.

### 3.1 Behavioural Abstraction

Suppose that  $\text{Spec}(a, b)$  is the *partial specification* of a device and  $\text{Imp}(a, b)$  is the design description. By partial specification we mean that  $\text{Spec}$  does not completely define the range of behaviour that the device can exhibit but only defines its behaviour in environments or states that are of particular interest.  $\text{Imp}$  is correct with respect to  $\text{Spec}$  if the following relation holds:

$$\forall a, b \cdot \text{Imp}(a, b) \Rightarrow \text{Spec}(a, b).$$

This asserts that whenever  $a$  and  $b$  satisfy  $\text{Imp}$  they will also satisfy  $\text{Spec}$ .

A device with input  $in$  and output  $out$  is specified by a predicate  $\text{Dev}$ , defined so that  $\text{Dev}(in, out)$  holds exactly when the combination of signals at  $in$  and  $out$  is one that is allowed to occur on the corresponding ports of the device. Modelling behaviours as relations on *signals*, that vary over time, makes the representation of state implicit. Thus, a delay with a propagation delay of  $\delta$  time units between input and output can be defined as

$$\text{Dev}(in, out) \hat{=} \forall t \cdot out(t + \delta) = in\ t.$$

Note that we say nothing about the output at times  $t < \delta$ . The system clock is *implicit* in this description. We view  $t$  as time measured from some implicit system reset. A suitable formalisation of a delay with implicit reset might be

$$\text{Dev}(in, out) \hat{=} \forall t \cdot out(t + \delta) = in\ t \wedge \forall t < \delta \cdot out\ t = lo$$

#### 3.1.1 Specifying the Components

The primitive components we use in the implementation are specified below, under the assumption of the existence of implicit clock and reset signals. The datatype `bool` consists of the set  $\{\text{hi}, \text{lo}\}$  and the operations `and`, `or`, `xor`, `not`, `implies` on values of `bool`.  $\mathcal{W}_4$  is a four-bit word.

```

val NOT(x,z) =  $\forall t:\text{time}.$  z t  $\equiv$  not(x t) ;
val AND(x,y,z) =  $\forall t:\text{time}.$  z t  $\equiv$  (x t) and (y t) ;
val OR(x,y,z) =  $\forall t:\text{time}.$  z t  $\equiv$  (x t) or (y t) ;
val XOR(x,y,z) =  $\forall t:\text{time}.$  z t  $\equiv$  (x t) xor (y t) ;
val COMP(inp1,inp2,z) =  $\forall t:\text{time}.$  z t  $\equiv$  (inp1 t = inp2 t) ;
val DELAY(x,z) =  $\forall t:\text{time}.$  z(0)  $\equiv$  lo  $\wedge$  z(t+1)  $\equiv$  (x t) ;
val LATCH(x,c,z) =  $\forall t:\text{time}.$  z(0)  $\equiv$  lo  $\wedge$ 
z(t+1)  $\equiv$  if (c t) then (x t) else (z t) ;

```

A 4-bit incrementer/register has a hold/increment input `inc:bool` signal, a reset input `clr:bool` signal and a 4-bit register `out: $\mathcal{W}_4$`  signal is specified below, where  $\oplus_{16}$  is binary addition of four-bit words, corresponding to addition modulo 16 of the integers they represent.

```

val INCR(inc,clr,out) =
  ∀ t:time. (not(clr t) ⇒
    out(t+1) = if inc t then (out t) ⊕4 1 else out t)
  ∧ (clr t ∧ not(inc t)) ⇒ out(t+1) = 04 ;

```

We do not specify what happens when `inc` and `clr` are both `hi`.

## 3.2 Structural Abstraction

Structural abstraction is expressed by *hiding* internal signals in the design description (using  $\exists$ ). Suppose an implementation  $\text{Imp}(x, y, z)$  consists of the interconnection of two simpler devices  $D_1(a, b)$  and  $D_2(c, d)$  with port  $b$  of  $D_1$  connected to port  $c$  of  $D_2$ . The behaviour of the resulting structure can be modelled by

$$\text{Imp}(x, y, z) \equiv D_1(x, y) \wedge D_2(y, z).$$

We might want to use  $\text{Imp}$  to implement a specification  $\text{Spec}(x, z)$  that constrains just the values of the external ports  $x$  and  $z$ . To relate  $\text{Imp}$  and  $\text{Spec}$ , we must abstract the structural information, that  $\text{Imp}$  has an ‘internal’ wire  $y$ . The abstracted behaviour can be modelled by

$$\text{AbsImp}(x, z) \equiv \exists y \cdot \text{Imp}(x, y, z)$$

that is, values  $x$  and  $z$  are allowed to occur on the external ports only when it is *possible* for internal signals to be generated such that  $\text{Imp}(x, y, z)$  holds.

### 3.2.1 Specifying Submodules

The point of our case study is to show we may represent, formally, common design practice — which includes top-down and bottom-up design. Designing top-down, we devise parts of the design concurrently with the refinement of the abstraction mappings that relate specification and implementation. Designing bottom-up we compose primitives to build components that match parts of our problem, and infer the abstractions from the implementation.

We *could* combine the formal, high-level specification of the stopclock with the abstraction mappings for its inputs and outputs, to calculate a low-level specification. The low-level specification is derived by expanding all high-level signals in terms of their concrete representation, thus achieving  $\text{Spec}(\text{abs } x)$ . The disadvantage of designing in this way is that structure inherent in the specification is not always carried over to the concrete level. Instead of dealing with the whole design at the implementation level, we show, here, how it may be partitioned into smaller problems *before* the refinement step.

**Top-down Design** For the stopclock, our specification suggests a partitioning into two subcircuits: one for producing `run` from the reset and start/stop signals; and the other (`Inc`) for storing and updating the counting of time. A block-level sketch of this ‘high-level’ design, corresponding to the partitioning implicit in the specification, can be made before further refinement – see Figure 1 – here, `Inc` and `Control` are

```

val Inc(reset,run,display) =
  ∀ τ:Utime. display(0) = {tens=0,secs=0,tenths=0} ∧
  display(τ+1) = if (reset τ) then {tens=0,secs=0,tenths=0}

```

```

else if (run  $\tau$ ) then nextTime(display  $\tau$ )
else (display  $\tau$ ) ;

val Control(reset,stst,run) =
   $\forall \tau$ :Utime. run(0) = lo  $\wedge$ 
  run( $\tau$ +1) = if (reset  $\tau$ ) then lo
               else if (stst  $\tau$ ) then not(run  $\tau$ )
               else (run  $\tau$ ) ;

```

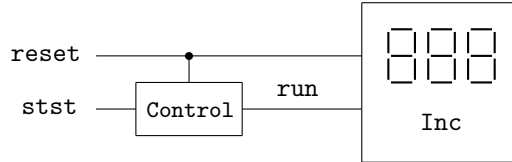


Figure 1: First partitioning.

Note that here we have only abstract, user-level, signals. Before introducing the lower level of time, we can further refine the implementation of `Control`. We may replace the conditionals by equivalent boolean logic, and introduce a delay to store the state. This allows us to “design” `CONTROL` as follows

```

val CONTROL(reset,stst,run)  $\equiv$   $\exists$  nextRun,resetBar,toggle:bool signal.
  DELAY(nextRun,run)  $\wedge$ 
  AND(resetBar,toggle,nextRun)  $\wedge$ 
  XOR(stst,run,toggle)  $\wedge$ 
  NOT(reset,resetBar) ;

```

We stress again that this ‘implementation’ is abstract. It forms the control subcircuit of the implementation shown in Figure 2. We will see soon how to refine it to a concrete level.

We can refine the design of `Inc` similarly (Figure 2). Here

```

val Nextn(reset,inc,carry,digit) =
   $\forall t$ :time.  $\exists$  digit t  $\Rightarrow$ 
    if reset t then (carry t = lo  $\wedge$  digit(t+1) = 0)
    else if inc t then
      if digit t = n then
        (digit(t+1) = 0  $\wedge$  carry t = hi)
      else
        (digit(t+1) = nextDigit(digit t)  $\wedge$  carry t = lo)
    else (digit(t+1) = digit t  $\wedge$  carry t = lo) ;

```

Note that we require the carries to be generated with zero delay.

### 3.3 Temporal Abstraction

Time is complex, partly because we have many different models of time, partly because the individual models may be complex, and, mostly, because we need to relate the different levels of timing abstraction, and the relation between different levels may be complex.

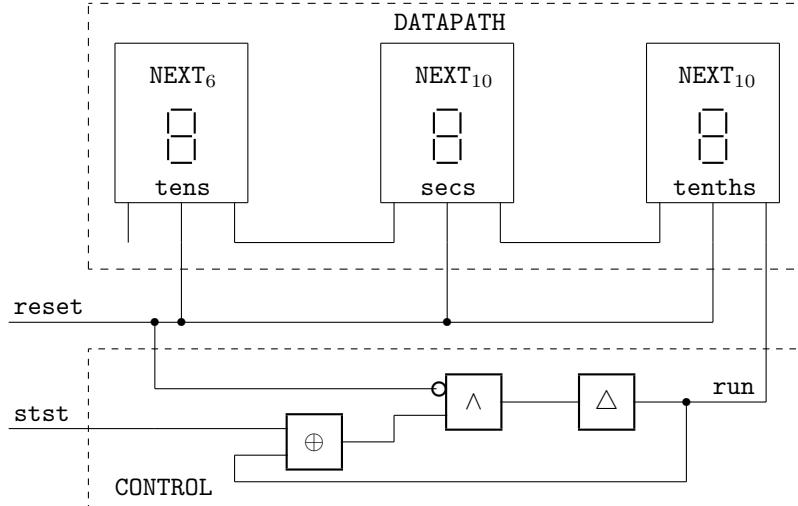


Figure 2: Second Partitioning.

Temporal abstractions pervade digital design; an assembly code view of a microprocessor may abstract from the realities of prefetch, pipelining and microinstructions, and view each instruction as atomic. Each time-step at this level is implemented in terms of a variable number of microinstructions. Each microinstruction step may, in turn, be implemented in some number of system clock cycles. Finally, various submodules may be clocked at different multiples of the system clock frequency. Of course, we could continue further in the other direction and consider the compilation of a statement in a high-level language to a number of machine instructions, and so on. Timing abstractions, in addition to data abstractions, are needed to relate these levels.

The simplest temporal abstraction is *fixed sampling*; let  $f$  be a monotone increasing function, mapping each abstract time  $t$  to a concrete time  $f t$ . Sampling of a signal is given by function composition (Melham *op. cit.*)  $Abs(x) \cong x \circ f$ . For example, slow-down [Leiserson86] can be modelled by the abstraction function  $\text{fn } t \Rightarrow n * t$ , i.e. the abstract sequence of values consists of every  $n^{\text{th}}$  value of the concrete sequence. Often, the sampling function is not fixed; the mapping has complex dependencies on the overall behaviour of the device; in mapping behaviour at microinstruction level to instruction level in a microprocessor, the sampling function depends on the program executed.

### 3.3.1 Temporal Abstractions for the Stopclock

For our case study, the abstract, user-level ticks model rising edges of the 10Hz clock. We introduce some useful functions to formalise the relation between this and the model that considers the signals clocked at 1MHz.

`rise x t` is true if `x` changes from `lo` to `hi` at `t`

```
fun rise x 0      = x 0
|   rise x (t+1) = x(t+1) and not(x t) ;
```

The recursive clause in this definition can be “translated” into a circuit in a straightforward way – see Figure 3. The initialization specified then follows.

```

val RISE(x,z) =  $\exists$  y,yBar:bool signal.  $\forall$  t:time.
                DELAY(x,y)  $\wedge$  NOT(y,yBar)  $\wedge$  AND(x,yBar,z) ;

```

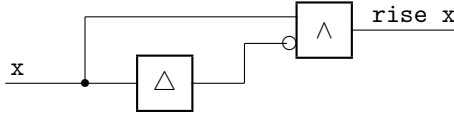


Figure 3: Implementing rise.

`next p t` returns the first time,  $t' \geq t$  at which  $p \ t'$  holds

```

fun next p t = if p t then t
               else next p (t+1) ;

```

`nth c n` returns the  $n$ th tick  $t'$  at which  $c \ t'$  holds:

```

fun nth c 0      = next c 0
  | nth c ( $\tau$ +1) = next c ((nth c  $\tau$ )+1) ;

```

Sometimes, we want to ignore the initial clock tick; `n+th c n` returns the  $(n+1)^{th}$  tick  $t'$  at which  $c \ t'$  holds:

```

fun n+th c  $\tau$  = nth c ( $\tau$ +1) ;

```

The fundamental sampling functions we require, to relate abstract to concrete times, are defined in terms of `tick`, i.e. the 10Hz clock signal: `smpl n` returns the low-level instant  $t'$  at which the  $n$ th rise of the 10Hz clock occurs.

```

val smpl = nth(rise tick) ;

```

`smpl+ n` returns the low-level instant  $t'$  at which the  $(n+1)^{th}$  rise of the 10Hz clock occurs.

```

val smpl+ = n+th(rise tick) ;

```

We introduce a name for `rise tick` since we will use this sampling signal frequently:

```

val s = rise tick ;

```

### 3.3.2 Changing Levels of Abstraction

We now refine the `Control` subcircuit to a physical level, where the signals vary with the 1MHz clock. This circuit will constrain the concrete signals, so that the abstract signals obtained by sampling them satisfy the constraints imposed by our earlier abstract decomposition. Because the connectives (conjunction and existential quantification) used to model composition of behaviours are monotone, this can be done component-wise, so we first consider how we might implement the abstract components.

**Combinational Logic** First note that, for a sampling abstraction, a concrete boolean logic gate implements the corresponding abstract logic gate. For example, recall the definition:

$$\text{AND}(x,y,z) = \forall t:\text{time}. z\ t \equiv (x\ t) \text{ and } (y\ t)$$

from this it follows that

$$\vdash \text{AND}(x,y,z) \Rightarrow \text{AND}(x \circ \text{smp1}, y \circ \text{smp1}, z \circ \text{smp1})$$

**Unit Delay** To implement an abstract delay, we use a concrete latch. Recall the definitions:

$$\text{DELAY}(x,z) = \forall t:\text{time}. z(0) \equiv \text{lo} \wedge z(t+1) \equiv (x\ t)$$

$$\text{LATCH}(x,c,z) = \forall t:\text{time}. z(0) \equiv \text{lo} \wedge z(t+1) \equiv \text{if } (c\ t) \text{ then } (x\ t) \text{ else } (z\ t)$$

To relate them simply we need to assume that the clock rises infinitely often. Then it is straightforward to show that

$$\text{INFOFTEN}(s) = \forall t:\text{time}. \exists t':\text{time}. t' \geq t \wedge s\ t' \equiv \text{true}$$

$$\text{INFOFTEN}(s) \vdash \text{LATCH}(x,s,z) \Rightarrow \text{DELAY}(x \circ \text{smp1}, z \circ \text{smp1})$$

If we represent the user-level signals, `reset`, `stst` and `run`, by sampling concrete signals, `RESET`, `SS` and `RUN` on `rise tick`, we can use these implementations of the abstract components to obtain a low-level implementation of `Control`.

This brings us to a crucial point; the abstract reset and start/stop signals are not given by sampling the concrete signals from the buttons. We will require *glue- or synchronization-logic* to match the abstraction relation relating the button and the abstract signal to the abstraction we use to implement `Control`. Since `run` is an internal signal, we can make sure we use the same abstraction at the interface to `Inc` so that no *internal* glue is necessary.

For example, the abstract signal `reset` will be defined as an abstraction of the concrete signal `resetButton`. Unfortunately, because *outputs must depend causally on inputs*, we will see that it is not possible to implement glue logic for the reset signal such that

$$\text{resetGlue}(\text{resetButton},s,\text{RESET}) \vdash \text{RESET} \circ \text{smp1} = \text{reset}$$

We cannot produce the correct value at the beginning of the low level interval, as the value may depend on later inputs. We *will* be able to satisfy

$$\text{resetGlue}(\text{resetButton},s,\text{RESET}) \vdash \text{RESET} \circ \text{smp1}^+ = \text{reset}$$

Since `smp1+` is again a sampling abstraction, this does not change our concrete implementation of the combinational logic, but we must take care to correctly implement the delay. The theorem we need is

$$\text{LATCH}(x,s,z) \wedge (x \circ \text{smp1})\ 0 = \text{lo} \vdash \text{DELAY}(x \circ \text{smp1}^+, z \circ \text{smp1}^+)$$

Note that there is an extra hypothesis. This gives us the following low-level implementation of `Control`

```

(NEXTRUN o smpl) 0 = lo  $\wedge$ 
reset = RESET o smpl+  $\wedge$  stst = SS o smpl+  $\wedge$  run = RUN o smpl+  $\wedge$ 
RISE(tick,s)  $\wedge$ 
LATCH(NEXTRUN,s,RUN)  $\wedge$ 
AND(RESETBAR,TOGGLE,NEXTRUN)  $\wedge$ 
XOR(SS,RUN,TOGGLE)  $\wedge$ 
NOT(RESET,RESETBAR)
 $\vdash$  Control(reset,stst,run)

```

where the first line is a proof obligation that we must eventually discharge. This will be possible once we have implemented the glue logic for the buttons.

### 3.3.3 Timing Abstractions for the Buttons

It may be argued that the abstraction functions we are about to define should have been part of the initial specification. For our case study, this may well be true; however, in ‘real life’ not *all* abstractions will be specified before design begins. In any case, we believe that it is helpful to *structure* specifications by factoring them into high-level specifications and abstraction functions.

**reset** The intention is that the reset button can be pressed asynchronously with respect to the user-level clock. Thus the abstract reset signal does not correspond to sampling the concrete signal produced by the button. The abstraction we use is

```

val some c x  $\tau$  =  $\exists t$ :time. nth c  $\tau$   $\leq$  t < nth c ( $\tau$ +1)  $\wedge$  x t  $\equiv$  hi ;

```

We define the abstract signal `reset = some s resetButton`, so

```

reset  $\tau$   $\equiv$   $\exists t$ :time. smpl  $\tau$   $\leq$  t < smpl( $\tau$ +1)  $\wedge$  resetButton t = hi

```

Now we see why it is not possible to represent `reset` by sampling an internal signal generated by glue logic on rising edges of the 10Hz clock; we cannot produce the correct value at the beginning of the low level interval, as the value depends on later inputs. However, if we define

```

fun Reset 0      = lo
|   Reset (t+1) = if s t then resetButton t else resetButton t or Reset t ;

```

then we *can* prove that

```

 $\vdash$  Reset o smpl+ = some s resetButton

```

We implement the recursive function directly – see Figure 4. Because this ensures that the signal `RESET` satisfies the equations determining the recursive function above, this gives us

```

resetGlue(resetButton,s,RESET)  $\vdash$  Reset o smpl+ = some s resetButton

```

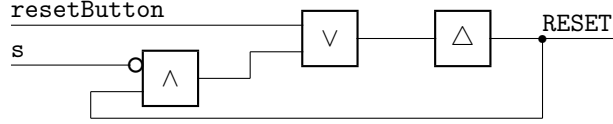


Figure 4: Glue logic for `reset`.

**start-stop** The abstraction we use for start-stop determines what will happen if the user presses the start-stop button a number of times during a  $1/10S$  interval. If this number is even, the stopclock should keep running normally; if the user presses the button an odd number of times within a  $1/10S$  interval, then the state (running/stopped) should change. The abstraction we use, where  $\#t \cdot \phi$  is the *number* of times  $t$  at which  $\phi$  holds, is

```
fun f n = case (n mod 2) of 0 => lo | 1 => hi ;
val tgl c x  $\tau$  = f  $\circ$  ( $\#t$ :time. nth c  $\tau \leq t <$  nth c ( $\tau+1$ )  $\wedge$  (rise x)t = hi) ;
```

We define the abstract signal `stst = tgl s ststButton` which counts the rises of the start-stop button. As we are counting the pressings modulo-2, a toggle behaviour is sufficient for keeping the count:

```
fun SS 0 = lo
| SS (t+1) = if s t then (rise ststButton)t
              else (rise ststButton)t xor SS t ;
```

This gives us

```
ststGlue(ststButton,s,SS)  $\vdash$  SS  $\circ$  smp1+ = tgl s ststButton
```

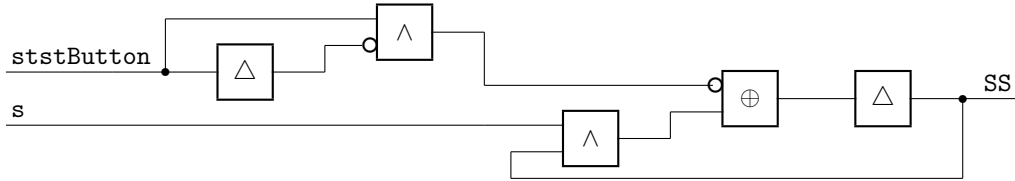


Figure 5: Glue logic for `SS` (start-stop).

Composing this circuit with our low-level implementation of `Control`, we can show

```
reset = some s resetButton  $\wedge$  stst = tgl s ststButton  $\wedge$ 
resetGlue(resetButton,s,RESET)  $\wedge$  ststGlue(ststButton,s,SS)  $\wedge$ 
RISE(tick,s)  $\wedge$ 
LATCH(NEXTRUN,s,RUN)  $\wedge$ 
AND(RESETBAR,TOGGLE,NEXTRUN)  $\wedge$ 
XOR(SS,RUN,TOGGLE)  $\wedge$ 
NOT(RESET,RESETBAR)
 $\vdash$  Control(reset,stst,run)  $\wedge$ 
  reset = Reset  $\circ$  smp1+  $\wedge$  stst = SS  $\circ$  smp1+  $\wedge$  run = Run  $\circ$  smp1+
```

where `Run` is given by

```

fun Run 0      = lo
| Run (t+1) = if RESET t then lo
              else (SS t) xor (Run t) ;

```

### 3.3.4 Timing Abstraction for the Display

When the clock is running, the value shown by the display “should change from one valid value to the next at each clock tick”. We *could* define

$$(\text{all } c \ x \ \tau = d) \equiv (\forall t:\text{time}. \ c \ \tau \leq t < c(\tau+1) \Rightarrow x \ t = d)$$

and use the abstraction `display = all smp1 DISPLAY`. However, this would not leave us much time to update the display and our ‘datapath’ would be expensive to implement. If we take advantage of the behaviour of the human eye, we can use several system-level clock cycles to change from one valid display to the next, without the eye perceiving intermediate invalid values. However, *most of the time*, the display must be correct. This idea can be captured by an abstraction function relating the high-level behaviour of the display to the interval of low-level time in which the picture displayed is a consistent value except for a few low-level ticks at the start of each  $1/10S$  cycle.

$$(\text{except } k_d \ c \ x \ \tau = d) \equiv (\forall t:\text{time}. \ (c \ \tau) + k_d \leq t < c(\tau+1) \Rightarrow x \ t = d)$$

The parameter,  $k_d$ , formalises the ‘few’ in the English reading. The choice of the constant  $k_d$  has to be delayed until we have a better idea of how the implementation will behave; its value will depend on the delays associated to the incrementing of the display. But we could, right now, impose a constraint on the implementation that  $k_d < 1000$ , say, and be sure that the resulting flicker in the display would be imperceptible to the user.

## 3.4 Data Abstraction

Specifications may use abstract datatypes, such as `Digit`; implementations, in current technology, use binary digits. Data abstractions relate the abstract data and its concrete representation. A simple, and common form of data abstraction is to view the data appearing in parallel on a bus as representing an abstract data object.

**Seven-segment Display** Our first example relates a seven-bit word to a single digit.

```

fun ssToDig w = case w of (hi,hi,hi,hi,hi,hi,lo) => 0
| ... |
| (hi,hi,hi,hi,lo,hi,hi) => 9 ;

```

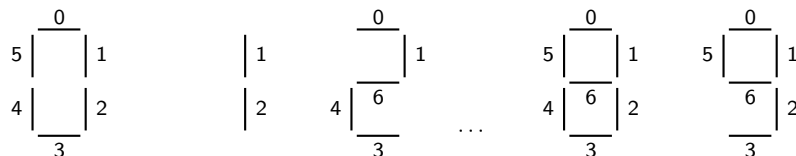


Figure 6: Encoding of digits in a seven-segments display

A seven-segment display implements this abstraction function

```
val SSD(inp,disp) =  $\forall t:\text{time}. \exists \text{ssToDig}(\text{inp } t) \Rightarrow \text{disp } t = \text{ssToDig}(\text{inp } t) ;$ 
```

This expression stands for “*if the input is in the appropriate range, then the value displayed corresponds to the number represented by inp*”. The encoding of the digits is shown in Figure 6.

**Decoder** Our incremter,  $\text{NEXT}_n\text{IMP}$ , uses a different representation of the digits

```
fun w4ToDig w = case w of (lo,lo,lo,lo) => 0
                        | ... |
                        (hi,lo,lo,hi) => 9 ;
```

The salient observation being that, with this abstraction,

```
 $\vdash \text{w4ToDig } w < 9 \Rightarrow (\text{nextDigit}(\text{w4ToDig } w) = \text{w4ToDig } (\text{incr } w))$ 
```

The decoder is just glue logic to relate the two data abstractions

```
val DECODER(inp,out) =
   $\forall t:\text{time}. \exists \text{w4ToDig}(\text{inp } t) \Rightarrow \text{ssToDig}(\text{out } t) = \text{w4ToDig}(\text{inp } t) ;$ 
```

Joining these two circuits, we obtain

```
val BINDISP(inp,dig) =  $\exists v:\mathcal{W}_7$  signal. DECODER(inp,v)  $\wedge$  SSD(v,dig) ;
```

which satisfies the theorem:

```
BINDISP(inp,dig)
 $\vdash \forall t:\text{time}. \exists \text{w4ToDig}(\text{inp } t) \Rightarrow \text{dig } t = \text{w4ToDig}(\text{inp } t)$ 
```

At the abstract level, we have just joined two lumps of wire to produce a wire.

### 3.5 Datapath

In this section, we implement  $\text{Inc}$ .

**Bottom-up Design** We use  $\text{INCR}$  and  $\text{COMP}$  to implement  $\text{Next}_n$ . We define  $\text{NEXT}_n\text{IMP}$  as follows

```
val NEXTnIMP(clear,inc,carry,word) =  $\exists z:\text{bool}$  signal.
  OR(clear,carry,z)  $\wedge$ 
  COMP(word,(Rep n  $\oplus_4$  1),carry)  $\wedge$ 
  INCR(inc,z,word) ;
```

then

```
NEXTnIMP(clear,inc,carry,word)
 $\vdash \forall t:\text{time}. \exists (\text{digit } t) \Rightarrow$ 
  if clear t then (carry(t+1) = lo  $\wedge$  word(t+1) = 0)
  else if inc t then if word t = (n + 1) then (word(t+1) = 0  $\wedge$  carry(t+1) = hi)
                    else (word(t+1) = (word t  $\oplus_4$  1)  $\wedge$  carry(t+1) = lo)
  else (word (t+1) = word t  $\wedge$  carry (t+1) = lo)
```

Here, `Rep` is the inverse of `w4ToDig`. If we use the right abstractions, `NEXTnIMP` implements `Nextn`. For the output, we want to combine the temporal abstraction `except kd smpl` with the data abstraction given by `w4ToDig`. For the control inputs and carry output we must beware! Our concrete component generates the carry after a unit delay. If we use a fixed sampling abstraction for the carry signals we fall foul of the delay in generating the carries. However, if we use different sampling abstractions for each carry signal then we can consistently maintain the abstract view; that the carries are generated with zero delay. We define

```
fun Δ-d x t = x (t + d) ;          for d ≥ 0
```

The theorem has some constraints:

```
∀t:time. CLEAR t = hi ⇒ ∃τ. smpl τ = t ∧
∀t:time. INC t = hi ⇒ ∃τ. smpl τ = t ∧
NEXTnIMP(CLEAR, INC, CARRY, WORD) ∧
digit = w4ToDig ◦ (except kd smpl WORD) ∧
clear = CLEAR ◦ smpl+ ∧
inc = (Δ-d INC) ◦ smpl+ ∧
carry = (Δ-(d+1) CARRY) ◦ smpl+
⊢ Nextn(clear, inc, carry, digit)
```

The first two lines represent a requirement that, if the abstraction is to hold, `CLEAR` and `INC` must only be `hi` at 10Hz clock ticks.

**INC** Finally, `Inc` is implemented with three instances of `NEXTnIMP`, three display-decoder pairs and some glue logic. `INC` takes as input the low-level versions of the reset signal `RESET`, the run/stopped signal `RUN`, the user-level signal `rise tick` and produces three “digits” as output `Dt`, `Ds` and `Dss`:

```
val INC(RESET, RUN, s, {tens, secs, tenths}) =
  ∃clr, inc, ct, cs, css:bool signal, xt, xs, xss:W4 signal. ∀t:time.
    AND(RESET, s, clr) ∧
    AND(RUN, s, inc) ∧
    NEXT9IMP(clr, inc, ct, xt) ∧
    NEXT9IMP(clr, ct, cs, xs) ∧
    NEXT5IMP(clr, cs, css, xss) ∧
    BINDISP(xt, tenths) ∧
    BINDISP(xs, secs) ∧
    BINDISP(xss, tens) ;
```

The datapath contains three subcircuits, one for each digit, each consisting of a 4-bit register/incrementer, a decoder for translating from a representation of numbers by 4-bit words into one in terms of 7-bit patterns and a 7-segment display for output. The `AND` gates are there so we can discharge the side-conditions, on `INC` and `CLEAR`, introduced above.

## 4 Putting It All Together

The sub-circuits comprising the stopclock will be timed by the 1MHz clock and yet behave as if synchronised by the 10Hz signal. This means, for instance, that the inputs to `INC` are 1MHz-level signals that change at the 10Hz ticks.



our present application, but it would be a bug. This bug would only appear if the user noticed that sometimes, on reset, the last value of the display was not maintained for long enough for him to read it. A similar bug in a safety-critical system, where the last value of the ‘display’ was used by another program for real-time safe shutdown, could be fatal. The appearance of this bug would depend on the timing of the shutdown reset signal, and it might well evade quite thorough simulation.

## 5 Caveat and Conclusions

The development sketched in this paper has *not* been mechanically verified; there are certainly bugs to be found, and details to be refined. Nevertheless, we believe it has some value. In any case, such a paper-and-pencil exercise is a prerequisite for a more rigorous machine-assisted synthesis.

**Acknowledgements** We have to acknowledge conflicting pressures — from Stuart Anderson who has penetratingly criticised several versions of this paper, suggesting major improvements in organisation, presentation and substance (which we have tried to implement), and from Graham Birtwistle who has persisted in asking for camera-ready copy long after others would have given up in despair. We thank them both for their patience and help.

We also thank Simon Finn, Michael Mendler and David Turner for pointing out several mistakes on the text and for their very helpful comments.

## References

- [AHLr90] Abstract Hardware Limited, *LAMBDA Reference Manuals Ver. 3.2*, 1991.
- [AHLu90] Abstract Hardware Limited, *LAMBDA User Guides Ver. 3.2*, 1991.
- [BG90] G Birtwistle, B Graham, *Verifying SECD in HOL*, in “Formal Methods for VLSI Design”, J Staunstrup (ed.), North-Holland, 1990.
- [BH90] B C Brock, W A Hunt Jr, *A Formal Introduction to a Simple HDL*, in “Formal Methods for VLSI Design”, J Staunstrup (ed.), North-Holland, 1990.
- [CGM86] A Camilleri, M Gordon, T Melham, *Hardware Verification Using High-Order Logic*, Univ. of Cambridge Computing Laboratory Tech. Rep. no.91, Sept. 1986.
- [Cohn87] A Cohn, *A Proof of Correctness of the Viper Microprocessor: The First Level*, Univ. of Cambridge Computing Laboratory Tech. Rep. no.104, 1987.
- [Cohn89] A Cohn, *Correctness Properties of the Viper Block Model: The Second Level*, in “Current Trends in Hardware Verification and Automated Theorem Proving”, G Birtwistle and G A Subrahmanyam (eds.), Springer-Verlag 1989.
- [FFFH90] S Finn, M P Fourman, M Francis, R Harris, *Formal System Design – Interactive Synthesis Based on Computer Assisted Formal Reasoning*, in “Formal VLSI Specification and Synthesis”, L J M Claesen (ed.), Elsevier Science Publishers, 1990.
- [FM89] M P Fourman, E Mayger, *Formally Based System Design – Interactive Hardware Scheduling*, in “Proc. of International Conference on VLSI”, G Musgrave and U Lauther (eds.), Munich, 1989.

- [Fourman77] M P Fourman, *The Logic of Topoi*, in “Handbook of Mathematical Logic”, Barwise (ed.), North-Holland, 1977.
- [Fourman86] M P Fourman, *Verification Using Higher-order Specifications*, in Proc. of the Silicon Design Conference, Wembley, 1986.
- [Fourman90] M P Fourman, *Formal System Design*, in “Formal Methods for VLSI Design”, J Staunstrup (ed.), North-Holland, 1990.
- [FPZ88] M P Fourman, W J Palmer, R M Zimmer, *Proof and Synthesis*, in Proceedings ICCD’88, Rye Brook, NY, 1988.
- [Gordon85] M Gordon, *Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware*, Univ. of Cambridge Computing Laboratory Tech. Report no.77, Sept. 1985
- [Gordon87] M Gordon, *A Proof Generating System for Higher-Order Logic*, Univ. of Cambridge Computing Laboratory Tech. Report no.103, Jan. 1987.
- [HLD89] F K Hanna et.al., *Formal Synthesis of Digital Systems*, in “Formal VLSI Specification and Synthesis”, L J M Claesen (ed.), Elsevier Science Publishers, 1990.
- [Herbert88] J Herbert, *Temporal Abstraction of Digital Designs*, Univ. of Cambridge Computing Laboratory Tech. Report no.122, Feb. 1988.
- [Hunt87] W A Hunt, *The Mechanical Verification of a Microprocessor Design*, in “From HDL Descriptions to Guaranteed Correct Circuit Designs”, D Borrione (ed.), North-Holland 1987.
- [Leiserson86] C E Leiserson, J B Saxe, *Retiming Synchronous Circuitry*, DEC SRC Report no.13, 1986.
- [Melham88] T Melham, *Abstraction Mechanisms for Hardware Verification*, in “VLSI Specification, Verification and Synthesis”, Proc. of the Workshop on Hardware Verification, Calgary, G M Birtwistle and P A Subrahmanyam (eds.), Kluwer Academic Press, 1988.
- [Mendler90] M Mendler, *Constrained Proofs: A Logic for Dealing with Behavioural Constraints in Formal Hardware Verification*, in “Designing Correct Circuits”, G Jones and M Sheeran (eds.), Oxford, Sept. 1990.
- [MTH90] R Milner, M Tofte, R Harper, *The Definition of ML*, MIT Press, 1990.
- [Weise89] D Weise, *Constraints, Abstraction and Verification*, in “MSI Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects”, M Leiser and G Brown (eds.), Springer Verlag, 1989.