Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, Proc. Formal Description Techniques VI, pages 71-86. North-Holland, Amsterdam, Netherlands, 1994.

DILL: Specifying Digital Logic in LOTOS

K. J. Turner[a], R. O. Sinnott[a]

[a] Department of Computing Science, University of Stirling, Stirling FK9 4LA, Scotland

As a relatively new application area for LOTOS (*Language Of Temporal Ordering Specification*), the specification of digital logic is investigated. A specification approach is evolved and justified, illustrated with basic logic gates and the larger example of a keyboard controller. The construction and validation of the digital component library are discussed, along with a retrieval tool that allows access to the library.

Keyword Codes: B.2.1; B.6.1; F.4.3

Keywords: Arithmetic and Logic Structures, Design Styles; Logic Design, Design Styles; Mathematical Logic and Formal Languages, Formal Languages; LOTOS (Language Of Temporal Ordering Specification)

1 Introduction

1.1 Applications of LOTOS

This paper addresses the specification and validation of digital logic components and circuits using LOTOS (*Language Of Temporal Ordering Specification*, [14]). LOTOS has been widely and successfully used to specify communications systems such as standards for OSI (*Open Systems Interconnection*). This is hardly surprising since LOTOS was developed for just this purpose. LOTOS has also been used to specify standards in the area of distributed systems such as the Trader of ODP (*Open Distributed Processing*, [15]) and TP (*Transaction Processing*, [26]).

However, LOTOS might claim to be a general-purpose language for applications that are sequential or concurrent, closely-coupled or distributed, communications or otherwise. The origins of LOTOS mean that it has so far seen little application outside OSI. Some forays have been made into related areas such as mobile communications [8], space communications [22], telephony [6] and CIM (*Computer Integrated Manufacturing*, [17]). However, completely new application areas are only now being sought. [10], for example, explores the value of LOTOS in the specification of artificial neural networks. This paper investigates how LOTOS can be used for the specification and analysis of digital logic designs. Other similar work on hardware description in LOTOS is reported in [7].

1.2 Digital Logic Specification in LOTOS

Digital logic design is well-understood; many textbooks (e.g. [16]) explain the operation of logic gates and how to combine them into larger circuits. Digital logic design is in practice constrained by the availability of specific hardware components that might be found in any manufacturer's catalogue. This allows standard components and circuits to be used.

Hardware description has been extensively studied. Languages such as CIRCAL, ELLA, HOL, LCF/LSM, RTL, temporal logic, VHDL and many others have been used to specify and

analyse hardware. The literature on this subject is vast; a few selected references are [3, 9, 11–13, 19, 20]. An interesting foil to the work reported in this paper is [24], which uses *occam* to specify and simulate logic circuits. In common with all such approaches, the goal of the work reported in this paper is to allow digital logic designs to be specified and analysed before actually building hardware. So why try to tackle tackle digital logic using LOTOS when there are already well established and successful approaches? There are several good reasons.

As a fully formal language, LOTOS supports rigorous specification and analysis in a way that semi-formal languages (e.g. VHDL) do not. The formal basis of LOTOS also allows full verification of designs. LOTOS inherits a well-developed theory of equivalences and relations from the field of process algebras. Hardware description languages with a similar origin (e.g. CIRCAL) have a similar advantage. However, because of its origin in data communications, LOTOS also has a well-developed theory of testing and test derivation (e.g. [2, 25]). This offers interesting alternatives to conventional hardware testing and simulation.

Some hardware description languages (e.g. ARCHI, MIDL) are intended for specification at the micro-architecture level only. Others (e.g. CDL, DDL, ISPS, RTL) are used at the register transfer level only. LOTOS can be used in a wide-spectrum manner at a number of levels of abstraction. This allows a consistent formalism to be used during hardware design, from the high-level architecture down to the gate or component level. Refinements between levels can be checked using standard LOTOS validation and verification approaches.

It is valuable to investigate the applicability of a language outside its original field (communications for LOTOS). New applications can help to discover the strengths and limitations of a language. LOTOS may be able to offer new insights and benefits compared to existing approaches. LOTOS is supported with tools that allow different analyses from those possible with other hardware description methods. Since the LOTOS philosophy is to write constructive specifications (though this is not enforced), simulation and rapid prototyping through tools offer attractive possibilities for hardware validation.

[23] shows how a 'component-based' style in LOTOS can be successfully used to specify digital logic designs. In particular, this approach faithfully reflects the view a hardware designer would take, allowing a clear correspondence between circuit designs and their LOTOS specifications. As noted by [7], a key ingredient in the successful use of LOTOS for hardware description is its support of multi-way synchronisation. Extensions to LOTOS have been proposed for specification of metric time and probability (e.g. [1, 18]). These extensions would allow timing (e.g. race conditions) and performance (e.g. reliability) to be specified and analysed formally in hardware design using LOTOS.

1.3 Overall Approach

Design is partly decompositional (top-down) and partly compositional (bottom-up). Engineering exploits this by aiming to use common components in different designs. Components are designed and manufactured to defined interfaces and standards, enabling them to be assembled with confidence into more elaborate structures. Another important aspect of engineering is that ready-made designs are often available. These combine known components in known ways to achieve predictable results.

Component re-use has been a major theme in software engineering for many years. However, in formal methods there has been little identification of useful specification components and specification structures using these. This is unfortunate since a major promise of formal methods

is verification of the system being specified. Verification is very hard for any but trivial systems, so verification of large or complex systems is usually infeasible in practice. A component-based style allows components to be verified individually. Larger combinations ('designs') of trusted components can then be verified more easily. A component-based style allows the specifier to take a higher-level, architectural view of the specification. This approach is elaborated in [23], where a component-based style for specifying digital logic is discussed. [21] gives a catalogue of logic components and designs in LOTOS. The earlier work of [7] takes a similar approach.

The purpose of this paper is to present the essential ideas of DILL (Digital Logic in LOTOS), as a complement to the work reported in [21, 23]. The DILL approach is explained in section 2, along with the logic component library that is currently supported. Section 3 discusses some fundamental issues for use of LOTOS to specify digital logic. As section 4 shows, even simple logic gates can be modelled in a number of different ways. A larger application of DILL in section 5 describes how a keyboard controller can be specified. Validation and verification issues are considered in section 6.

2 The DILL Approach

This section explains the philosophy and elements of the DILL approach.

2.1 Philosophy

The basic philosophy of DILL is that it should be easy for the hardware engineer to translate a circuit schematic into a LOTOS specification, and then to analyse and verify the properties of this specification. Once the specification is considered to be correct, it should be practicable to realise it using the chosen hardware technology. This approach has several implications for DILL.

There is a need for a component library – a collection of LOTOS specification fragments that describe commonly available components. To ease the designer's task, the component library should match the hardware technology that will be used. For example, the CMOS chips supplied by a particular manufacturer might be the target. The library should contain generic definitions of components that can be instantiated in a specification. Since components are specified in LOTOS, the designer must be familiar with how to combine LOTOS behaviour expressions; fortunately the rules are reasonably straightforward and do not require an in-depth knowledge of LOTOS.

It should be easy to reference the specifications of components in the library. This requires documentation of the library, and a tool to extract the required specification fragments. The designer should not need to know the internal construction of the library. In particular, if a component is built from simpler ones, then their definitions should be included automatically. The library tool should also avoid multiple copies of a component specification, as might arise if a simpler component is used in several larger components that are needed.

It should be possible to describe logic designs at different levels of abstraction, and to refine one level into a more detailed design. DILL does not provide guidelines for this, since refinements will be motivated by normal hardware design procedures. However, DILL – through LOTOS – supports wide-spectrum specification at a variety of levels, and has a well-developed theory of equivalences and relations between specifications.

It should be possible to analyse and verify the specification generated from a design. Happily,

there is a wealth of LOTOS tools to help here, so this aspect does not require specific DILL support. Existing LOTOS tools support a variety of analyses. Simulation can be used to check behaviour of a circuit manually. Unfolding behaviour to a certain depth might be used to detect deadlocks. Checking equivalences and relations between specifications could verify correctness of a lower-level design with respect to a high-level one. Specifications might be analysed for behavioural properties such as safety or liveness.

It should be possible to realise a specification generated by DILL fairly directly in hardware. This is straightforward in the sense that DILL mirrors a conventional hardware design. However, an exciting possibility that has not (yet) been explored would be to design a 'silicon compiler' for LOTOS. Various LOTOS compilers already translate low-level LOTOS (with annotations) into a programming language such as C or Ada. In principle, a compiler could also be designed that would turn LOTOS into, say, a mask to be etched onto silicon or links to be set in a PLA (*Programmable Logic Array*).

The key elements in DILL are thus the component library and the associated retrieval tool. A theory of refinement and general LOTOS tools already exist. Only a silicon compiler might be needed as a special development.

2.2 Library Components

A preliminary component library has been produced for DILL. This is constructed in a bottom-up fashion, from basic gates to more complex components. The internal structure of composite components is hidden, so their behaviour is observationally equivalent to the desired behaviour and their construction is hidden from the user. The component library should be oriented towards particular chip sets, but has not yet been. Instead the emphasis has been on defining commonly available components. Validation of the library is discussed in section 6.

The current library is summarised in figure 1; the structure shown is for convenience in presentation only. The names of components are derived from hardware design practice. The library has some fundamental components (e.g. a generic one-input logic gate) that are unlikely to be useful to a designer and so are not shown. Components with a repeated structure have been supplied only for limited cases (e.g. a two-input decoder, a two-stage shift register). There would be no difficulty in generalising these components for some size *n* (e.g. a 2^n-input decoder, an *n*-stage shift register).

2.3 Accessing the Library

The use of a component from the DILL library is declared by giving its name and the suffix '**Decl**'. The documentation needed to use the library is little more than an elaboration of figure 1. Most components have unparameterised declarations (e.g. **Inverter_Decl** for inverters). A few have parameters describing their function (e.g. **Gate_Decl**(nand,3) for three-input *nand* gates). All components are specified as LOTOS process abstractions, though there are some supporting data type definitions. An instance of a component is thus a LOTOS process instantiation. A complete specification is generated with appropriate 'rubric' using the declaration:

circuit(*specification_parameters*,*specification_behaviour*)

In fact, DILL is a rather thin veneer on top of LOTOS, so the specification parameters and behaviour are given directly in LOTOS. Component specifications are included automatically by declaring them after the behaviour, but it is up to the designer to say how the components are wired up. This is done by combining the components in a LOTOS behaviour expression,

Component Type	Name	Purpose
Basic Logic	One	Source of logic 1
	Sink	Sink of logic signal
	Zero	Source of logic 0
1-Input Gate	Delay1, Delay2	Repeat input after 1, 2 unit delay
	Inverter	Complement input
	Repeater	Repeat input
2-Input Gate	And2	Binary *and*
	Nand2	Binary *nand*
	Nor2	Binary *nor*
	Or2	Binary *or*
	Xor2	Binary *xor*
3-Input Gate	And3, ...	Ternary *and*, ...
4-Input Gate	And4, ...	Quaternary *and*, ...
Coder	Decoder2	Decode 2 bits as one of 4 outputs
	Encoder2	Code one of 4 inputs as 2 bits
Plexer	Demultiplexer2	Select one output using 2-bit code
	Multiplexer2	Select one input using 2-bit code
Adder	HalfAdder	Sum and carry of 2 bits
	FullAdder	Sum and carry of 2 bits and previous carry
	ParallelAdder2	Parallel sum of 2 bit pairs
	RippleAdder2	Ripple-through sum of 2 bit pairs
Latch	CDRSLatch	(Re-)settable, two-input, clocked bistable
	CRSLatch	(Re-)settable, clocked bistable
	DLatch	One-input, clocked bistable
	DRSLatch	(Re-)settable, two-input bistable
	RSLatch	(Re-)settable bistable
Flip-Flop	DFlipFlop	One-input, clocked memory
	JKFlipFlop	Reliable, two-input, clocked memory
	MSFlipFlop	Two-input, clocked memory
	TFlipFlop	Divide-by-2 counter
Counter	Clock	Source of clock pulses
	Divider2	Divide-by-2 counter
	Divider4	Divide-by-4 counter
	Divider8	Divide-by-8 counter
Register	BucketBrigade2	Two-stage bit sequence repeater
	PassOn2	Two-stage event sequencer
	ShiftRegister2	Two-stage bit pattern shifter

Figure 1. DILL Component Library

synchronising on the gates that correspond to connections. Internal connections are specified as hidden gates.

A larger application of DILL is given in section 5. As a simple illustration for now, consider a circuit design to carry out an *andnot* function – a binary *and* where one input is inverted. Its DILL declaration might be:

```
circuit(
  'AndNot2 [Ip1, Ip2, Op]','
    hide NotIp2 in And2 [Ip1, NotIp2, Op] |[NotIp2]| Inverter [Ip2, NotIp2]
  where
    And2_Decl
    Inverter_Decl
')
```

A DILL description is run through a pre-processor to extract the required component declarations and produce a LOTOS specification. The pre-processor requires commas in arguments to be quoted, which explains the syntax above. For the above description, the generated specification would have the following form; the specification wrapping and the component declarations are generated automatically. The auxiliary processes *Logic1* and *Logic2* are discussed in section 4.

```
specification AndNot2 [Ip1, Ip2, Op] : noexit
  library ...
  type ...
  behaviour
    hide NotIp2 in And2 [Ip1, NotIp2, Op] |[NotIp2]| Inverter [Ip2, NotIp2]
  where
  process Logic2 ...
  process And2 ...
  process Logic1 ...
  process Inverter ...
endspec (* AndNot2 *)
```

DILL is supported by a library of macros written in *m4* – a widely available macro processor that runs on Unix and other systems. The macros are merely a convenient means of parameterising and generating LOTOS text for each kind of logic gate or component. The library contains about 70 macros in 900 lines of *m4* to specify the components mentioned in this paper.

3 Digital Logic Specification Style

When specifications have to be written in a new application area, it is common to find that a significant amount of experimentation is needed to discover the best approach. For specifying digital logic, the authors evaluated several approaches before finding a satisfactory style. In particular, it turned out that the way in which even simple gates were modelled was critical to combining them into larger circuits. This is especially true of circuits with feedback (such as flip-flops). It is easy to introduce inadvertent deadlock[1] if the specified components do not quite fit together properly, even though they seem to behave correctly in isolation. See [21] for examples of the kinds of problems that arise with an incorrect approach.

[1]Real hardware deadlocks if it enters a state in which subsequent inputs have no effect. Livelock occurs when hardware indefinitely processes internal signals without reacting to external ones.

3.1 Modelling Digital Signals

In reality, digital signals take on a range of analogue values (e.g. from 0 to 5 volts) but thresholds are set so that they may be treated as logic 0 or 1. As a signal changes from one value to another, it may pass through an indeterminate state that is neither logic 0 nor 1. It might therefore seem that tri-state logic should be used, allowing for an 'undefined' state of signals. This, however, would make specifications much more complex. An undefined signal level should always be transient and therefore should be ignored if possible. As a workable abstraction, signals are regarded as having only two states.

Logic design proceeds on the basis of binary signals. However, as an implementation matter, there is a choice of how logic 0 and 1 correspond to electrical signals. Normally 0/1 corresponds to low/high, called positive logic. However, negative logic may also be used in which 0/1 corresponds to high/low. This is an implementation decision that depends on the components available. Either approach can be used with the same logic design. Since this is essentially an implementation matter it is ignored in a functional specification, but is considered in a lower level specification.

There is also a choice of whether a signal (a level) or a change in signal (an edge) should be modelled as a LOTOS event. Choosing to model signal levels means that a gate must repeatedly offer its current output value in events since a signal level is continuous. LOTOS events are discrete, so a level can be represented only as a succession of arbitrarily close events giving the signal value. Specifying this would confuse the behaviour with identical repeated events. It is therefore more satisfactory to model signal *changes* as LOTOS events.

Given that signal changes should be modelled, there is still a choice in LOTOS. Events could simply indicate a change without saying whether this was from 0 to 1 or vice versa. This would not be a good reflection of reality, where the direction of a change is explicit. The overall conclusion is that LOTOS events should indicate the direction of a change by giving the newly established level (e.g. *g ! 1* for a transition from 0 to 1).

Open circuits have to be allowed for. For example, an input may be left floating, a component input may not be used, or an unused output may not be attached to anything. A floating input corresponds to some default value. Signals on an unused input are simply absorbed. Floating outputs produce signals, but they go nowhere; in LOTOS terms, these are hidden internal events.

3.2 Modelling Logic Gates

3.2.1 Reflecting Real Gates

Logic functions (logic gates[2]) are the basic components of digital logic. Logic functions could perhaps be modelled as ADT (*Abstract Data Type*) operations on input values. However, the time-dependent behaviour of logic circuits is often important, so it is better to use LOTOS behaviour expressions. More importantly, a specification using ADTs would not readily support 'wiring up' a circuit. Each logic gate is therefore specified as a LOTOS process, instantiated with appropriate parameters.

A real logic gate exhibits a propagation delay from a change in input to the subsequent output. This appears naturally in a LOTOS specification since output events follow input events. However, the actual time delay between such events is not modelled in LOTOS. For many purposes the exact delay is unimportant, since a design that assumed specific propagation delays in each real gate might be prone to race conditions. Many logic designs are synchronous to

[2]Since 'gate' has both a hardware meaning and a LOTOS meaning, the term is qualified where necessary.

avoid such problems, and this removes the need to model delays explicitly. If it were necessary to quantify the propagation delays, one of several timed variants of LOTOS such as [1, 18] could be used. Such an approach might be justified for asynchronous logic or for investigating race conditions.

Real gates are connected by wires from outputs to inputs. The wires (should) accurately transmit signals, but they can introduce a propagation delay that is critical in high-speed circuits. The wires could be considered as components as well, but to do so would make specifications very unwieldy. In virtually all logic designs the wires can be ignored, but where their effect is significant then they can be specified as delays. Ignoring the wires makes connection of components very easy in LOTOS: events at connected output and input gates are synchronised by giving them the same LOTOS gate name. In effect, a gate name is given to a wire. Multi-way synchronisation in LOTOS also allows one output to be sent to several inputs[3].

Real gates have a fan-out (the maximum number of other gates that can be connected to an output). This is an implementation restriction that is best ignored in a specification (though a static analysis could perhaps determine whether fan-out limits have been complied with). Real gates also have a fan-in (the number of inputs) that is intrinsic and should be specified.

In real life, and in a specification, switching on a circuit leads to a certain amount of settling down. Faster hardware gates will produce their outputs earlier, and this may determine the stable state (especially if the circuit has feedback). However, the circuit should enter a stable state after a short time. The specification of such a circuit will behave similarly, with internal events initially until the behaviour settles down. Since exact propagation delays are not specified in standard LOTOS, there may be non-determinism in which stable state is reached – as with real hardware.

Switching off a circuit is likely to result in complex behaviour as signals and power decay; such behaviour is unlikely to be interesting. The effect of a clean switch-off could be modelled simply as disruption in LOTOS terms. If a tidy hardware power-down were required, the circuitry would be specifically designed for it; the LOTOS specification would therefore be written to match.

3.2.2 Gate Inputs and Outputs

The decision to model signal changes as LOTOS events means that logic gates must remember the previous state of inputs. This is realistic since a hardware gate will be in some state, with current flowing or not. The previous levels on each input are therefore parameters of the process that models a logic gate. Default values must be supplied when the process is instantiated, corresponding to the state of a hardware gate when switched on. Typically, inputs will initially act as if 0, but this may be gate-dependent. Since the defaults may depend on the specific gate, they are given in its definition rather than as parameters of the process that is instantiated.

There is a little subtlety in specifying when output should occur. In practice, a logic gate reacts to a change in just one of its inputs. The specification of a logic gate should therefore reflect this and should not, for example, require all input events to occur before producing an output. The specification of a logic gate must also not force the output of a new value after an input changes. In circuits involving feedback (e.g. a flip-flop), requiring output before further input could lead to deadlock. In practice, there may be a short input pulse (say, from 0 to 1 to 0) to which a gate cannot react quickly enough. Since real gates have a propagation delay, an

[3]Of course, no attempt should be made to synchronise two outputs, any more than the output of two real gates should be connected.

input pulse of short duration may not produce an output. Allowing a further input before output is therefore both realistic and necessary.

An important corollary of modelling only signal changes means that a new input to a logic gate may not produce a new output, though initially a logic gate may produce output. Consider, for example, an *and* gate with 0 and 0 as initial inputs; it may at first produce an output event with value 0. If one input subsequently changes to 1, the output will remain at 0 and there will be no output event. A short input pulse may not result in output as described above. All these considerations mean that a logic gate must know its previous output as well as its previous inputs.

3.2.3 Parameterising Gates

LOTOS offers more possibilities for dealing with inputs and outputs than are found in real hardware. An obvious approach is to make each input and output correspond to a LOTOS gate. This might be termed 'physical multiplexing', because each LOTOS gate corresponds to a physical connection (e.g. events *g3 ! 0* and *g4 ! 1*). LOTOS also allows what might be described as 'logical multiplexing', in which a LOTOS gate is qualified by a connection number parameter in events (e.g. events *g ! 3 ! 0* and *g ! 4 ! 1*). The advantage of the second approach is that a LOTOS gate may then correspond to an arbitrary number of inputs or outputs. This does not faithfully reflect real logic gates, however, which are built with a fixed number of connections. Also it considerably complicates how the wiring up of components is specified. With one gate per input or output, wiring up merely requires inputs and outputs to be synchronised. With gates qualified by connection numbers, a 'master configuration process' would be needed to synchronise on all events and allow only connected outputs and inputs to communicate. Physical multiplexing is therefore preferable in the interests of simplicity and realism.

Although gates with any number of inputs above two are possible, preferred numbers of inputs (e.g. 3, 4, 8) tend to be usual; unused inputs can be wired to logic 0 or 1 as appropriate to make them ineffective. LOTOS could allow a logic gate process have a parameterised number of inputs by making use of logical multiplexing, but as argued above this is undesirable. A fixed number of inputs and outputs is therefore specified, each being a LOTOS gate.

Hardware gates are designed to implement a fixed function[4]. Although each kind of gate could be separately specified in full, this would lead to a lot of duplication since the behaviour of a gate is largely separate from its actual logic function. LOTOS can be more flexible by allowing generic logic gate definitions, parameterised with their function. Such an approach appears to break from a strict representation of real gates. However, the instantiation of a process is akin to the fabrication of a real gate. Just as fabrication fixes the function of a real gate, so does instantiation of a logic gate process.

Because LOTOS does not allow operations to be given as parameters to processes, their *names* rather than the operations themselves are supplied. An *Apply* operation is used to calculate the result of a logic function from its name and operands. For example, *Apply (nor, 1, 0)* yields 0. The specification of ADTs for logic functions is straightforward and is not considered further in this paper.

[4] A ULA (*Uncommitted Logic Array*), PLA (*Programmable Logic Array*) or CLA (*Configurable Logic Array*) might be considered as an exception.

4 Basic Logic Gates

This section discusses how basic logic signals and logic gates are specified.

4.1 Logic Sources and Sinks

Sometimes it is necessary to specify a source of logic 0 or 1, say to tie an input to a specific level. This is a nullary logic function, specified by the process *Source* that outputs its parameter once (since the signal never changes). Processes *Zero* and *One* provide logic 0 and 1 by instantiating *Source*:

```
process Source [Op] (BOp : Bit) : noexit :=
  Op ! BOp;  stop
endproc (* Source *)
```

It may also be necessary to absorb an input signal without using it:

```
process Sink [Ip] : noexit :=
  Ip ? b : Bit;  Sink [Ip]
endproc (* Sink *)
```

4.2 One-Input Gate

A one-input gate can play one of two different roles: as a repeater (amplifier, delay) or as an inverter. The corresponding logic functions are *same* and *not*. The approach discussed in section 3 leads to a surprisingly complex specification of a generic one-input gate:

```
process Logic1 [Ip, Op] (BOp : BitOp) : noexit :=
  Logic1A [Ip, Op] (BOp, 0 of Bit)
  where
  process Logic1A [Ip, Op] (BOp : BitOp, BIn : Bit) : noexit :=
    Ip ? BInNew : Bit;  Logic1A [Ip, Op] (BOp, BInNew)
  []
    (  let BOutNew : Bit = Apply (BOp, BIn) in
       Op ! BOutNew;  Logic1B [Ip, Op] (BOp, BIn, BOutNew)  )
  endproc (* Logic1A *)
  process Logic1B [Ip, Op] (BOp : BitOp, BIn, BOut : Bit) : noexit :=
    Ip ? BInNew : Bit;  Logic1B [Ip, Op] (BOp, BInNew, BOut)
  []
    (  let BOutNew : Bit = Apply (BOp, BIn) in
       Op ! BOutNew [BOutNew ne BOut];  Logic1B [Ip, Op] (BOp, BIn, BOutNew)  )
  endproc (* Logic1B *)
endproc (* Logic1 *)
```

This logic gate process is parameterised by a unary logic function and has a default input value of 0. Initially it may input values and output the resulting logic value. But once it has output a value, it may output only if the value changes. Although this specification is complex for good reasons, the specifier can treat it as a black box and not be concerned with its details. As a concrete example of a one-input logic gate, an inverter has the specification:

```
process Inverter [Ip, Op] : noexit :=
  Logic1 [Ip, Op] (not)
endproc (* Inverter *)
```

4.3 Two-Input Gate

A two-input gate can perform one of 16 different logic functions. Only some of these are usually given names such as *and*, *or* (inclusive or), *xor* (exclusive or). Certain logic functions are convenient to implement in hardware, so *nand* and *nor* are also common. Names could be given to the other binary logic functions, but would rarely be needed and would be unlikely to correspond to real gates. A generic two-input gate is specified much as a one-input gate, and is parameterised with the name of a binary logic function. Because of its similarity to the one-input gate, only an outline specification is given:

```
process Logic2 [Ip1, Ip2, Op] (BOp : BitOp) : noexit :=
  Logic2A [Ip1, Ip2, Op] (BOp, 0 of Bit, 0 of Bit)
  where
  process Logic2A [Ip1, Ip2, Op] (BOp : BitOp, BIn1, BIn2 : Bit) : noexit := ...
  process Logic2B [Ip1, Ip2, Op] (BOp : BitOp, BIn1, BIn2, BOut : Bit) : noexit := ...
endproc (* Logic2 *)
```

For this logic gate, there are two inputs with default value 0. As a concrete example of a two-input gate, a *nor* gate has the specification:

```
process Nor2 [Ip1, Ip2, Op] : noexit :=
  Logic2 [Ip1, Ip2, Op] (nor)
endproc (* Nor2 *)
```

4.4 Higher-Level Components

Some logic gates can be built in other ways. For example, a *nand* gate could be built from an *and* gate feeding into an inverter. The gate might actually be built this way, but the availability of *nand* gates in practice means that it is realistic to specify them directly. However, the *andnot* gate described in section 2.3 is not a normal hardware component, and so is specified compositely. Logic gates with more than two inputs (e.g. a four-input *and* gate) can be specified like simpler gates, but using *n*-ary Boolean operations. Three-input and four-input gates have been specified in the course of this work.

More complex components can be built progressively out of basic logic gates. [23] gives some representative examples, while [21] presents a catalogue of the component specifications corresponding to figure 1. The next section shows some of these components at work in a larger example.

5 Designing a Keyboard Controller

This section presents a larger application of DILL for specifying a keyboard controller.

5.1 Controller Design

A computer keyboard is usually a matrix of switches that are operated by pressing keys. The keyboard is associated with a keyboard controller that deals with low-level hardware aspects and presents a simple interface to the CPU. For this example, the CPU treats the keyboard as a device that can be periodically polled to find out which key (if any) is currently pressed. (In a more sophisticated design the keyboard controller could interrupt the processor on a key press.) A computer engineering reference book might suggest the keyboard design shown in figure 2.

The keyboard controller does not include the keyboard matrix – a rectangular matrix whose intersections contain keyboard switches. For this example, a small 4×4 keyboard is assumed

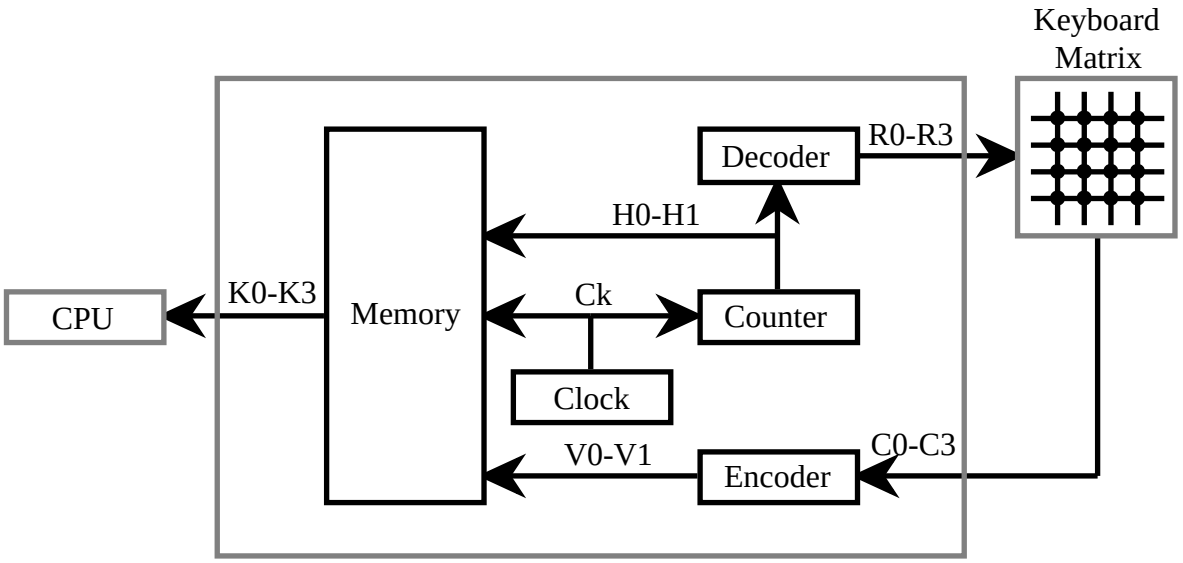

Figure 2. Keyboard Controller Design

(say, a numeric keypad). In most cases there are parallel connections between the controller components, carrying a number of bits. The connection names are given in figure 2: *C* – column, *Ck* – clock, *H* – horizontal, *K* – key, *R* – row, *V* – vertical. The components in the keyboard controller are as follows.

Clock produces alternate 0 and 1 signals at a rate which is unimportant in this example; in practice, the rate would have to be much faster than the expected rate of key presses. *Counter* cycles through a sequence of binary values: 00 to 11 in this example since the matrix has four rows. *Decoder* produces a 1 signal on the output that is uniquely determined by its binary input; a four-from-two decoder is needed. As an example, input of 10 to *Decoder* would result in a 1 signal on *R2*. *Encoder* reverses the action of *Decoder*, producing the binary code corresponding to the input that is set to 1; a two-from-four encoder is needed since the matrix has four columns[5]. As an example, a 1 signal on *C1* to *Encoder* would result in an output of 01. *Memory* is a four-bit memory that stores the row and column numbers of the currently pressed key. Values are periodically clocked in, and may be read out independently by the CPU. The sixteen keys are identified by a four-bit number, two bits giving the row and two bits giving the column.

5.2 DILL Representation

The target chip set is likely to include components corresponding directly to *Clock*, *Counter*, *Decoder* and *Encoder*; in DILL terms these are *Clock*, *Divider4*, *Decoder2* and *Encoder2* components. The specification of *Memory* can be put to one side for the moment, allowing the current design to be declared in DILL as:

circuit(
‘KeyCon [C0, C1, C2, C3, K0, K1, K2, K3, R0, R1, R2, R3]’,‘

[5]To guard against keys in different columns being pressed simultaneously or no keys being pressed at all, this should be a priority encoder; however, the simple encoder in the DILL library has been used for this example.

```
    hide Ck, H0, H1, V0, V1 in
      (
        Encoder2 [C0, C1, C2, C3, V0, V1]
      |||
        (
          Decoder2 [H0, H1, R0, R1, R2, R3]
        |[H0, H1]|
          (
            Divider4 [Ck, H0, H1]
          |[Ck]|
            Clock [Ck]
          )
        )
      )
    |[Ck, H0, H1, V0, V1]|
      Memory [Ck, H0, H1, V0, V1, K0, K1, K2, K3]
  where
  Encoder2_Decl
  Decoder2_Decl
  Divider4_Decl
  Clock_Decl
  process Memory [Ck, H0, H1, V0, V1, K0, K1, K2, K3] : noexit := ...
')
```

Memory is likely to be built from four instances of *DFlipFlop* (a standard one-bit memory element), all sharing a common clock input. The negated output of each flip-flop is not required and can be hidden. The DILL declaration for the whole design can therefore be completed with the following specification of *Memory*:

```
hide NotK0, NotK1, NotK2, NotK3 in
  DFlipFlop [H0, Ck, K0, NotK0] |[Ck]| DFlipFlop [H1, Ck, K1, NotK1]
|[Ck]|
  DFlipFlop [V0, Ck, K2, NotK2] |[Ck]| DFlipFlop [V1, Ck, K3, NotK3]
```

The specification for the keyboard controller is generated by running the DILL description through the pre-processor. The resulting LOTOS text is about 290 lines (including blank lines) and is not given here for space reasons. Ideally, the designer will not need to even read the specification, and should be able to simulate or analyse it directly. However, as the next section shows there are practical difficulties in handling even examples of this size.

6 Validation and Verification

The specification of every DILL library component has been checked in considerable detail by simulation with the *SMILE* tool [5]. This required each component to be synchronised with a test harness to drive it. Although the specifications use full LOTOS, the events are of rather a simple form: a gate with parameter 0 or 1. It might therefore have been possible to determine the canonical tester automatically using results from LOTOS testing theory [25]. Validation consists of stepping the specification of the component and test harness through every significant path. For larger components, this is a manageable but tedious operation.

The LOTOS simulators available to the authors (*SMILE* and *hippo*) caused difficulties with internal events, especially when simulating larger specifications. For example, a JK flip-flop at switch-on presents 12 internal events that should be allowed to occur before the first external input event. This is a reflection of what actually happens with the hardware, but it places a heavy overhead on the simulation. The internal events represent spontaneous establishment of internal signal levels. Components with feedback (such as flip-flops) 'race' to establish one of the possible stable states. Other flurries of internal events occur after an input signal.

When the whole specification of the keyboard controller in section 5 is simulated, the initial event menu presents 81 events of which 69 are internal! The internal events must be allowed to happen first so that the circuit can settle down. After initial settling down, the observable events that are possible are: strobing a keyboard matrix row; reading out a signal on a keyboard matrix column; and reading the current state of the key memory. After each observable event, there may be further internal events as new signal levels are communicated and new stable states are reached via transient unstable ones. The number of internal events after the initial settling is much less, but it still places a burden on the user of the simulator. It requires nearly 100 simulator event selections to go through the following behaviour: initial settling of the keyboard controller; strobing the keyboard matrix once; registering the key pressed; reading out the number of the current key; and advancing the clock, ready for the next strobe. Each subsequent clock tick needs roughly 10 simulator event selections.

What is clearly required is a means to control the simulator as far as internal events are occurred. Most internal events are uninteresting, and should be allowed to happen in any order until only observable events have to be selected for simulation. In fact, only one internal event in the keyboard controller specification is of any interest for simulation purposes: the internal clock tick. In real life – and in an ideal simulation – the other internal events will occur in a spontaneous and irrelevant fashion after each observable event. Another problem with the simulators is that internal events in replicated components are presented with the same name in the event menu, making precise selection difficult. Simulator limitations are a difficulty for the DILL approach, though the tools rather than DILL are at fault. Simulator developers have already recognised the problem of event control. For example, a future version of SMILE [4] will incorporate a Tool Control Language; among other things this will allow automatic selection of internal events until a new stable state is reached.

The ultimate objective of this work is to have a fully *verified* library of components that can be used in the design of arbitrarily complex logic systems. Verification of components by exhaustive testing is possible in principle, though it is very time-consuming with complex circuits. Other approaches such as model-checking may be attractive alternatives. As larger circuits are considered, it becomes harder to see when two designs are equivalent; the same behaviour may be obtained from two designs that at first sight are rather different. This is an obvious role for equivalences in LOTOS, particularly observational congruence and testing congruence. Although tools that check LOTOS equivalences generally work only on basic LOTOS, the simple event structure offers hope of either enhancing the tools or translating the full LOTOS specifications into basic LOTOS.

Hardware engineers often use techniques for minimisation of logic designs in order to reduce the number of components needed. Equivalence checking would be useful to verify that the simplified design conforms to the original design. Certain kinds of simplification in LOTOS (e.g. parameterised expansion) might be relevant to carrying out minimisation directly.

7 Conclusions

A style for specifying digital logic in LOTOS has been introduced and justified. This has been used to specify a variety of logic gates and larger components. A particularly pleasing outcome is that components can be specified and analysed in isolation, and can be easily built into larger combinations. However, such a component-based style depends critically on specifying components that fit together properly. The weakest aspect of the work at present is the difficulty of checking components by simulation, though planned simulator improvements will help greatly.

Future work required on digital logic specification in LOTOS includes investigation of test derivation, equivalence checking, model checking, and LOTOS timing and probability extensions for analysis of performance issues. DILL currently lacks a facility to declare arrays of components, though this would not be difficult to achieve.

If LOTOS were to become a serious competitor to established hardware description languages, much more work would be necessary. A wider range of case studies including rather larger examples would be needed. A more comprehensive library would be essential. Efficient simulation and theorem-proving capabilities would have to be developed. Nonetheless, it is felt that the present level of this work shows promise in the field of hardware description.

Acknowledgements

R. O. Sinnott was supported by the UK Science and Engineering Research Council during the work described in this paper. The authors thank Dr. R. G. Clark for comments, and the referees for their valuable suggestions.

References

1. Bolognesi, T. and Lucidi, F.: 'LOTOS-Like Process Algebras with Urgent or Timed Interactions', Proc. *Formal Description Techniques* IV, pp. 249–264, North-Holland, Amsterdam, NL, 1991.
2. Brinksma, E.: 'A Theory for the Derivation of Tests', Proc. *Protocol Specification, Testing, and Verification* VIII, pp. 63–74, North-Holland, Amsterdam, NL, 1989.
3. Cohn, A.: 'The Notion of Proof in Hardware Verification', J. of *Automated Reasoning*, 5 *(2)*, pp. 127–139, 1989.
4. Eertink, H.: Private communication regarding future developments in SMILE, University of Twente, NL, 1993.
5. Eertink, H. and Wolz, D.: 'Symbolic Execution of LOTOS Specifications', Proc. *Formal Description Techniques* V, North-Holland, Amsterdam, NL, 1992.
6. Faci, M., Logrippo, L. and Stépien, B.: 'Formal Specification of Telephone Systems in LOTOS', Proc. *Protocol Specification, Testing, and Verification* IX, pp. 25–34, North-Holland, Amsterdam, NL, 1989.
7. Faci, M. and Logrippo, L.: 'Specifying Hardware in LOTOS', Proc. *Computer Hardware Description Languages and their Applications* XI, pp. 305–312, North-Holland, Amsterdam, NL, 1993.

8. Fernández, A. *et al.*: 'PRODAT – The Derivation of an Implementation from its LOTOS Formal Specification', Proc. *Protocol Specification, Testing, and Verification* VIII, North-Holland, Amsterdam, NL, 1988.
9. Filkorn, T.: 'A Method for Symbolic Verification of Synchronous Circuits', Proc. *Computer Hardware Description Languages and their Applications* IX, pp. 229–239, North-Holland, Amsterdam, NL, 1991.
10. Gibson, J. P.: 'A LOTOS-Based Approach to Neural Network Specification', Technical Report 112, Department of Computing Science, University of Stirling, UK, 1993.
11. Gordon, M. J. C.: *HOL: A Proof Generating System for Higher-Order Logic*, Technical Report 103, University of Cambridge, UK, 1987.
12. Hunt, W. A.: *The Mechanical Verification of a Microprocessor Design*, Technical Report CLI-6, Computational Logic Inc., Austin, USA, 1987.
13. IEEE: *VLSI Hardware Design Language*, Project 1076, Institute of Electrical and Electronic Engineers, New York, USA, 1987.
14. ISO: *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, ISO 8807, International Organisation for Standardisation, Geneva, CH, 1989.
15. ISO: *Information Processing Systems – Open Distributed Processing – Formal Description of the Trader*, ISO JTC1/SC21/WG7/N743 Annex G, International Organisation for Standardisation, Geneva, CH, 1992.
16. Kline, R. M.: *Structured Digital Design*, Prentice-Hall, USA, 1983.
17. McClenaghan, A.: 'Experience of using LOTOS within the CIM-OSA Project', Proc. *Formal Description Techniques* IV, pp. 109–116, North-Holland, Amsterdam, NL, 1991.
18. McClenaghan, A.: 'Distributed Systems: Architecture-Driven Specification using Extended LOTOS', Ph.D. Thesis, University of Stirling, 1993.
19. Milne, G. J.: 'The Formal Description and Verification of Hardware Timing', IEEE Trans. on Computers, 40 *(7)*, pp. 811-826, 1991.
20. Moskowski, B.: 'A Temporal Logic for Multi-Level Reasoning about Hardware', Proc. *Computer Hardware Description Languages and their Applications* VI, North-Holland, Amsterdam, NL, 1983.
21. Sinnott, R. O.: *The Formally Specifying in LOTOS of Electronic Components*, M.Sc. Thesis, University of Stirling, UK, 1993.
22. Taylor, C. R. and Gamble, M: 'The CCSDS Protocol Validation Programme Inter-Agency Testing using LOTOS', Proc. *Formal Description Techniques* III, North-Holland, Amsterdam, NL, 1990.
23. Turner, K. J.: 'An Engineering Approach to Formal Methods', Proc. *Protocol Specification, Testing, and Verification* XIII, North-Holland, Amsterdam, NL, 1993 *(in press)*.
24. Welch, P. H.: 'Emulating Digital Logic using Transputer Networks', *Parallel Architectures and Languages Europe*, LNCS 258, pp. 357–373, Springer-Verlag, Berlin, D, 1987.
25. Wezeman, C. D.: 'The CO-OP Method for Compositional Derivation of Conformance Testers', Proc. *Protocol Specification, Testing, and Verification* IX, North-Holland, Amsterdam, NL, 1989.
26. Widya, I., van der Heijden, G.-J. and Juillot, F.: 'Specification and Implementation of the TP Protocol', Lo/WP3/T3.1/N0077/V04, ESPRIT Project 2304, Commission of the European Communities, Brussels, 1992.