Kenneth J. Turner, F. Javier Argul Marin and Stephen D. Laing. Concurrent Specification and Timing Analysis of Digital Hardware using SDL (extended version of article published by and copyright Springer-Verlag). In Jose Rolim et al., editors, Proc. International Parallel and Distributed Processing Symposium, Cancun, Mexico, LNCS 1800, pages 1001-1008, Spring Verlag, Berlin, Germany, May 2000.

Concurrent Specification And Timing Analysis of Digital Hardware using SDL

Kenneth J. Turner, F. Javier Argul-Marin and Stephen D. Laing

Computing Science and Mathematics, University of Stirling
kjt@cs.stir.ac.uk, javargul@bbvnet.com, stephenl@reading.sgi.com

Abstract

Digital hardware is treated as a collection of interacting parallel components. This permits the use of a standard formal technique for specification and analysis of circuit designs. The ANISEED method (Analysis In SDL Enhancing Electronic Design) is presented for specifying and analysing timing characteristics of hardware designs using SDL (Specification and Description Language). A signal carries a binary value and an optional time-stamp. Components and circuit designs are instances of block types in library packages. The library contains specifications of typical components in single/multi-bit and untimed/timed forms. Timing may be specified at an abstract, behavioural or structural level. Timing properties are investigated using an SDL simulator or validator. Consistency of temporal and functional aspects may be assessed between designs at different levels of detail. Timing characteristics of a design may also be inferred from validator traces. A variety of examples is used, ranging from a simple gate specification to realistic examples drawn from a standard hardware verification benchmark.

Keywords: Concurrent System, Hardware Description, SDL (Specification and Description Language), Timing Specification

1 Introduction

1.1 Background

Digital hardware can be viewed as a concurrent system whose components operate in parallel but synchronised via exchange of electrical signals. Although SDL (Specification and Description Language [15]) was developed for specifying communications systems, it is a general-purpose language of wide applicability. It is the contention of this paper that SDL is appropriate and useful for specifying and analysing digital hardware as collections of interacting parallel components. The approach particularly focuses on timing aspects, which are often tricky in hardware design.

HDLs (Hardware Description Languages) were initially developed only as descriptive tools, but they were soon associated with formal methods. Much of the literature on formal methods for hardware design appears in the proceedings of CHDL (Computer Hardware Description Languages and their Applications, e.g. [8]).

SDL is of interest to hardware specifiers because it offers rigorous specification, good system structuring features, high-level communication, and the possibility of hardware-software co-design. In these respects it complements industrial hardware description languages such as

VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language [13]) and VERILOG [14].

Most uses of SDL for hardware description have aimed at synthesis using standard engineering tools. As reported in [2, 7, 9, 19, 20, 21], SDL hardware descriptions are often translated into VHDL. This allows SDL to be used for high-level hardware description, coupled with common tools for hardware synthesis and further analysis. Hardware-software co-design using SDL has also been investigated [10, 17, 18]. Hardware elements are usually generated via VHDL, while software elements are generated via C or similar. SDL toolsets that support co-design include COSMOS [6] and ODE [11].

1.2 Goals

The authors are engaged in the project ANISEED (Analysis In SDL Enhancing Electronic Design [1, 5]). Its goals are complementary to those of others who have used SDL for hardware description. Specifically, translation to VHDL and/or C is assumed to be dealt with by other tools. Instead, the authors have concentrated on timing aspects of hardware specification and analysis. The goal is to allow timing constraints on circuits and components to be specified and analysed at various levels: *abstract* (overall sequencing constraints), *behavioural* (black-box viewpoint), and *structural* (internal design). As well as being the project name, ANISEED also refers to the hardware description method and the special-purpose tools.

Hardware engineering tends to focus on design. As a result it deals with relatively low-level issues. This also means that specification and design are rather close. In *software* engineering, a much sharper separation is made between requirements, specification and design. ANISEED brings this perspective to hardware engineering by using SDL in the early stages of requirements definition and specification. The aim of ANISEED is therefore to model a system before it is realised as even a hardware prototype. This higher-level, software-inspired approach allows the feasibility and characteristics of a circuit to be evaluated at an early stage.

Section 2 describes the overall approach to specifying/analysing circuits and their timing characteristics using SDL. The use of validation and verification for SDL is discussed in section 3 Section 4 explains how SDL can be used to specify abstract timing constraints of various standard forms. The paper then presents a graded series of examples to illustrate the approach. Of necessity as the level of complexity in the examples rises, the amount of detail that can be given in the paper falls. More information is available in a separate report [1]. As a small but instructive example, section 2.4 shows how functionality and timing of a *nand* gate can be specified. A more complex component appears in section 5, which describes a D (Delay) flip-flop. Section 6 shows a simple circuit, the Single Pulser drawn from a catalogue of standard hardware verification benchmarks [23]. Section 7 shows a much more complex circuit, the Bus Arbiter that is another standard hardware verification benchmark.

2 Approach

The behaviour of hardware components can be modelled naturally using SDL processes, since these run in parallel and communicate via signals. The communication model of SDL requires processes to receive inputs from a queue and is thus asynchronous, unlike real hardware. It is still, however, possible to use SDL for both synchronous (clocked) and asynchronous (unclocked) logic. SDL is appropriate for specifying hardware timing since it supports metric time. This is not necessarily real time since the passage of time is under the control of a scheduler. For timing analysis, ANISEED can use a standard SDL simulator or validator. However, the authors have also implemented a discrete event simulation by automatically modifying the scheduling strategy of a standard SDL simulator. This gives more flexibility in the way that timing analysis is performed.

2.1 Signal Specification

A hardware signal is modelled as an SDL signal with time-stamp (optional) and value parameters:

Time-stamp optionally records the time at which a signal is considered to have been generated. This is necessary partly because an output signal may not be consumed immediately; it still, however, carries the time of its generation. If SDL timers cause output only when required by the timing parameters, a standard simulator can be used. However, as discussed in section 3.2 a time-stamp is useful when a signal is generated in advance of its proper time.

Value is mandatory and may simply be a bit. However, in general it may be multi-bit (i.e. a list of bits). This is appropriate at a high level of specification where a bus or group of wires is to be specified as a whole. For example, a 32-bit register is conventionally regarded as having a 32-bit inputs and outputs rather than 32 individual wires. In a very abstract specification it might even be desirable to carry arbitrary values such as data structures. ANISEED supports uni-bit and multi-bit signals, such that a multi-bit higher-level specification may be related to a uni-bit lower-level specification. Although binary signals have the value of 0 or 1, a bit variable is also permitted to have the value *X* (meaning unknown). This is necessary when defining the initial state of a circuit on startup. The bit operators have to deal with *X* values (e.g. 1 *or X* is 1, 1 *and X* is *X*).

ANISEED allows uni-bit/multi-bit and untimed/timed specifications. Library components are available in all four bit/time combinations, with variants being automatically generated. The specifier can thus choose low-level (uni-bit) or high-level (multi-bit) models for interconnection, and can choose to omit or include timing characteristics. Untimed specifications are simply a special case of timed ones: only the relative ordering of events is specified, so a time-stamp is omitted from signals. It is useful to write an untimed specification first in order to check the functional correctness of a design. Timing constraints can then be added (by a change of library component names), allowing timing issues such as race conditions and hazards to be studied.

2.2 Component Specification

The electronics designer can choose from a wide range of components in various families. These range from simple elements such as logic gates (*and*, *or*, etc.), through intermediate components like multiplexers and flip-flops, to complex components like registers and parallel adders. ANISEED is therefore supported by a library of common components and circuits (i.e. designs consisting of a number of components.) These are stored in SDL packages, forming a modular and easily extended library. Some of the packages depend on others (for example, all packages use bits).

A type of component is modelled as a block type in SDL. A block type is a (parameterised) block definition that can be statically instantiated to yield a particular block; an example appears in section 2.4. The motivation for choosing block types rather than process types is mainly that the internal construction of a component should be invisible. The specifier should not need to know if the component contains one process or many interconnected processes. This also means that circuit designs as well as black-box components can be stored in the library. A further consideration is that library block types are instantiated statically, better matching the notion of using a specific component. A block type is parameterised by its signal names and its gates (in the SDL sense). Timed components are also parameterised by characteristics such as their propagation delay or setup time.

Some 'components' in the ANISEED library may not quite correspond to real hardware elements. For example, a source of logic 0 or 1 is a pseudo-component in the library; in practice it would be a connection to the circuit ground or supply level. For high-level timing constraints, pseudo-components are available to define various interrelationships among signals. These are used only in the initial stages of design, and are later replaced by specific components.

Components are interconnected by no-delay channels. Like real wires these are considered to convey signals instantaneously. If it is necessary to model the propagation delay of a wire, as in high-speed circuits, a delay component can be used. A limitation of SDL is that an output cannot be broadcast to arbitrary processes. To solve this problem, ANISEED uses junction 'components' that model where wires connect. Although these appear in a circuit diagram as small blobs, the specifier must instantiate a junction block type to link the components.

Where multi-bit components are interconnected with uni-bit components (e.g. a 4-bit adder feeding into four inverters), a split 'component' is used to separate the bits. Correspondingly a merge 'component' is used to combine uni-bit signals into a multi-bit signal.

2.3 ANISEED Library

It would have been possible to specify all the library components individually. However this would have been very tedious. For example, a two-input *nand* gate has largely the same specification as one with three, four or eight inputs. The gates for *and*, *or*, *nor* (not-or), *xor* (exclusive-or) and *xnor* (exclusive-not-or) differ from *nand* only in their logic function. Since each kind of logic gate has uni-bit/multi-bit and untimed/timed versions, a total of $4 \times 6 \times 2 \times 2$ (input×function×bit×timing) or 96 variants would have to be specified explicitly.

As a more pragmatic solution, all variants are generated automatically from an SDL template that is parameterised by the logic function, the number of inputs, whether timed and whether multi-bit. The template is an outline PR (SDL Phrasal Representation) specification that is pre-processed to yield the required variants. Although the macro facility of SDL was investigated for this, it is not sufficiently flexible. Instead the *m4* macro processor [22] is used. The *m4* library modules are automatically pre-processed to generate the PR library packages. The approach using templates makes the *m4* library much smaller (10%–15% in size compared to the generated PR). The approach makes the library more maintainable, since a single template needs to be changed if the model of a component is changed. This is also important since different models may be used for different purposes (e.g. validation as opposed to synthesis, asynchronous as opposed to synchronous design). A simple change of macro parameter can

Package	Components
arith	arithmetic units such as adders
bit1	uni-bit values and operators
bitm	multi-bit values and operators
coder	decoders (binary input to outputs), encoders (inputs to binary output)
flipflop	flip-flops (simple memory elements)
gate	logic gates, logic sinks, logic sources
junction	connections of wires
latch	latches (simple storage elements that do not decouple output from input)
merge	multiple uni-bit inputs to single multi-bit output
mux	demultiplexers (one input, many outputs), multiplexers (many inputs, one output)
seq	abstract sequencing constraints
split	single multi-bit input to multiple uni-bit outputs
tristate	tri-state devices (output disconnected when disabled, e.g. for connection to a bus)

Figure 1: ANISEED Library Packages

select an appropriate model and also timing characteristics for a particular family of logic gates.

The current ANISEED library is summarised in figure 1. It contains over 400 standard components such as might be found in a typical logic family. The average size of each library component specification is about 80 SDL PR lines, varying from 9 to 300 lines. The library components have all been verified, with an average of about 300 states per component. A number of other library packages are currently under development.

Explanatory comments are automatically generated from the *m4* templates when SDL PR is produced. The library packages are thus quite human-readable. Since GR (SDL Graphical Representation) is often preferred by SDL specifiers, the library templates can also automatically generate comments in the style of CIF (Common Interchange Format [16]). This allows an SDL tool to produce an acceptable graphical representation of the library PR components. (The default graphical representation produced by SDL tools when converting PR is often not very readable since graphical layout is a tricky task.) Most specifications in this paper are given as PR. This is partly because the library is textual anyway, and partly because it is more convenient to present the specifications with comments.

2.4 A Simple Component: A *Nand* Gate

As a trivial example to illustrate the modelling approach, a two-input timed *nand* (not-and) gate has input signals *SIp1/SIp0* and output signal *SOp*. These carry time and bit values. Hardware components typically have different output delays *TDel1* and *TDel0* for outputting a 1 or a 0. Timing parameters depend on the family of hardware being modelled, e.g. CMOS (Complementary Metal Oxide Semiconductor) or ECL (Emitter-Coupled Logic). According to the specifier's choice, a particular set of delay values is imported. Hypothetical values might be:

```
synonym CMOSDel1 = 5;                              /* 1 output delay */
synonym CMOSDel0 = 4;                              /* 0 output delay */
```

The ANISEED library specification of the *nand* gate is given below. Delay and signal names are given as context parameters. Signal lists for gates and signalroutes are implicit. The *nand* gate accepts inputs at any time and stores them. Operator *Apply1* calculates the output from a named logical operation and its parameters. (The '*1*' suffix indicates that it acts on single bits; there are also multi-bit '*M*' operators.) Operator *NewOut1* decides if the output must change as a result of new input. (If the inputs change from 0,0 to 0,1 a *nand* gate does not need to output since the previous value of 1 is still valid.) A timer is used to delay the output according to its value. Any further input in the meantime causes the output calculation to start again. In hardware description terminology this is called pure delay: when an input dictates a new output value, this will appear after the delay. ANISEED also allows specification of inertial delay: the component does not respond to input changes that are too short (i.e. brief pulses).

The main complication is what happens when the *nand* gate first powers up. With real hardware, power-up results in an arbitrary output until this can be properly determined from the inputs. If the *nand* has received no input or just one input then its output may be random. For example if a *nand* gate has received a 1 on just one input, the state of its output is not yet determined. Operator *AnyOut1* decides whether any output is permitted, i.e. whether a 1 or 0 should be chosen non-deterministically. The reason for outputting something in these circumstances arises when a component is used in sequential logic. Such designs have feedback so that outputs feed into earlier inputs. If a component did not output until all its inputs had been received, there would be deadlock.

```
block type Nand2T <                                                   /* timed 2-input nand */
  synonym TDel1, TDel0 Duration;                          /* context parameters for timing ... */
  signal SIp1 (Time, Bit1), SIp0 (Time, Bit1), SOp (Time, Bit1);>                 /* and signals */
  gate Ip1 in; gate Ip0 in; gate Op out;                                /* input/output gates */

  process Nand2T (1, 1);                                              /* one process instance */
    signalset SIp1, SIp0;                                                    /* input signals */
    dcl BIp1, BIp0, BOp, BNextOp Bit1 := X;             /* input/output values start unknown */
    dcl TIp, Top Time;                                         /* last input/next output time */
    dcl TDel Duration;                                                      /* required delay */
    timer T;                                                                   /* delay timer */
    start;                                                              /* component power-up */
      nextstate Ready;                                                /* now ready for input */
    state Ready;                                                          /* ready for input */
      input SIp1 (TIp, BIp1), SIp0 (TIp, BIp0);                            /* get either input */
        nextstate –;                                                 /* ready for more input */
      provided NewOut1 (NandB, BIp1, BIp0, BOp);                          /* output to change? */
        task BNextOp := Apply1 (NandB, BIp1, BIp0);                           /* set next output */
        decision BNextOp;                                                       /* decide delay */
          (1): task TDel := TDel1;                                            /* 1 output delay */
          (0): task TDel := TDel0;                                            /* 0 output delay */
        enddecision;                                                          /* delay now set */
        task TOp := TIp + TDel;                                              /* set output time */
        set (now + TDel, T); nextstate Waiting;                               /* wait for delay */
      provided AnyOut1 (NandB, BIp1, BIp0, BOp);                           /* any output OK? */
        decision any;                                                       /* random 1 or 0 */
          (): task BOp := 1;                                                      /* choose 1 */
          (): task BOp := 0;                                                      /* choose 0 */
        enddecision;                                                /* random output now set */
        output SOp (0, BOp); nextstate –;           /* output at time 0, ready for more input */
    state Waiting;                                                            /* wait for delay */
      input SIp1 (TIp, BIp1), SIp0 (TIp, BIp0);                          /* get a new input */
        reset (T); nextstate Ready;                       /* cancel delay, ready for more input */
      input T;                                                             /* delay expired */
        task BOp := BNextOp; output SOp (TOp, BOp);        /* output result at required time */
        nextstate Ready;                                             /* ready for more input */
  endprocess Nand2T;

endblock type Nand2T;
```

2.5 Overcoming Tool Restrictions

The authors use version 3.5 of the TAU/SDT toolset [24]. This has some restrictions that affect its suitability for Aniseed. The axioms of data types (the bit types in the Aniseed library) are ignored by the SDT simulator and validator. This is understandable since it is difficult to compile axioms into efficient code. Instead, SDT allows data type operators to be defined as procedure-like SDL operators or directly in C. Unfortunately SDT does not permit the former to be used in continuous signals (which Aniseed requires). The *m4* library modules for bits therefore generate two PR variants: one has axioms, and the other has C code for operators.

A more severe problem is that commercial SDL tools (SDT, ObjectGeode) do not currently support context parameters fully. These are essential for block types since the actual timing parameters and signal names are not known until a block type is instantiated in a particular context. Aniseed allows use of SDL context parameters as normal. To overcome tool limitations, Aniseed automatically instantiates types in the PR generated from a graphical description. For example, an instance of the timed *nand* gate discussed in section 2.4:

```
block SomeNand : Nand2T <CmosDel1, CmosDel0, Ip1, Ip0, Op>;
```

is translated into a block definition with context parameters substituted:

```
block SomeNand;
  process Nand2T (1, 1);
    ... input Ip1 (TIp, BIp1), Ip0 (TIp, BIp0) ...
    ... task TDel := CmosDel1 ... task TDel := CmosDel0 ...
    ... output Op (TOp, BOp) ...
  endprocess SomeNand;
endblock SomeNand;
```

3 Validation and Verification

3.1 Checking SDL

SDL tends to be used in pragmatic ways, so validation is usually the method of choice. What would normally be termed verification (proof, model-checking) is comparatively rare for SDL [3, 12].

The SDL community uses the term validation to mean automated checking of an SDL specification. Validation is used to check for undesirable conditions such as unreachable states, unspecified receptions, deadlocks and process input queues growing without bounds (a symptom of livelock). A specification may be validated in isolation; such a check is useful but does not confirm functional correctness. Alternatively a specification may be validated against an MSC (Message Sequence Chart [4]). The MSC may be written by the specifier, in which case the validation amounts to testing. The MSC may also be derived from an earlier validation run. This can be used for regression testing, i.e. to check whether a revised specification respects the same behaviour as previously. More usefully, the MSC may be derived automatically from a higher-level specification. The MSC then contains all the behaviour found at the higher level, and can thus be used to confirm that a lower-level specification is a correct refinement. The confidence level in this kind of validation depends on the completeness of the higher-level MSC.

A typical SDL validator like SDT offers a number of validation strategies. In the case of exhaustive validation, all states and paths of the specification are followed. Successful validation of this kind leads to a complete MSC that can be used to verify correctness of a refinement. The SDT validator can carry out exhaustive analysis of a typical library component in about a minute, requiring some hundreds of states. For full circuit designs, exhaustive analysis becomes computationally infeasible.

Instead, the SDT random-walk validation is employed for realistic circuits. This proceeds multiple times from a given starting point to a given depth, making random choices where there is a branch in the state space. Although this does not guarantee complete exploration of the state space, it is very effective. By running the validator several times, hundreds of thousands of states can be checked in a matter of minutes. The validator has settings (notably the search depth) that can be adjusted to achieve 100% symbol coverage after a number or runs. This gives confidence in the validation even though random-walk validation is not technically exhaustive.

3.2 Checking Timing Characteristics

Most SDL validation is oriented towards checking functional correctness. However, SDL allows timing aspects to be specified and thus validated. In ANISEED, interactive simulation can be used to check timing behaviour according to the tester's expectations. Simulation is time-consuming since it is driven manually. Instead, automated validation is preferred. The SDT validator can carry out exhaustive analysis of a typical library component in about a minute, but random-walk validation is the norm for typical circuit specifications.

MSCs are a convenient graphical means of showing how hardware components interact. However, the MSCs resulting from validating realistic circuits tend to be lengthy and hard to follow. This is partly because of the large number of internal signals, and partly because of the large number of interactions. The authors therefore developed a tool that converts MSCs into the more conventional timing diagrams used by electronics engineers. An example appears later in figure 5.

For hierarchical timing specifications, the validator is useful in checking consistency between different design levels. This is an important point in real-world hardware design, since complex circuits are commonly designed in a top-down fashion. High-level functional units are progressively broken down to the level of available components. There is a risk of introducing an error during refinement of a complex design. With the ANISEED approach, errors show up as inconsistencies in timing or functionality between the different design levels. In such a case, an MSC trace at the higher level will not be accepted when validating the lower level. Due to state space explosion, it is usually not practicable to compare two specifications at widely differing levels of abstraction. However, it is feasible to check the refinement of consecutive levels in the design process. For example the abstract and behavioural specifications may be compared, or the behavioural and structural specifications. Since the design steps are then smaller, the MSCs and specifications are more comparable.

The MSC traces produced by the validator are also useful in deriving timing characteristics. Manufacturing tolerances and environmental differences mean that two 'identical' components rarely have the same timing characteristics. ANISEED can therefore be used to give a range of values for each timing parameter. When many components are interconnected, the resulting validator traces can be used to determine the range of high-level timing properties. As an example, the 1 and 0 output delays may be slightly different for each logic gate. In a complex circuit it may not be obvious how these variations will interact to produce overall timing characteristics. The validator output for a variety of traces can be analysed to determine the statistical bounds on these values. In effect the validator can be used for Monte-Carlo simulation and analysis. That is, many (automated) validator runs generate statistical timing information from which bounds on timing characteristics can be derived.

A standard SDL simulation deals with signals in the order that they are generated. For timing simulation this may be incorrect, since a signal should be considered only at the time given by its time-stamp. This situation can arise in two circumstances: when simulation inputs are not in the desired time order, and during automated validation. It is convenient for someone testing a circuit to define a test scenario in a human-oriented way (e.g. a truth table for combinational logic or a transition table for sequential logic). In such a case, test inputs may not be provided in correct time order. The other possibility for misordering arises during automated validation. The SDT validator does not advance time during validation, so the time-sequencing normally guaranteed by SDL timers is not applicable. In such a case, events must be consumed in order of their time stamps.

The signal with the earliest time-stamp must be consumed first, even if other signals have been placed before it in the input queue of a process. The normal procedure for interpreting SDL may therefore need to be modified. When a signal is added to an input queue, ANISEED can be configured to store it according to the time-stamps of the signals. The usual FIFO scheduling algorithm then selects signals in the correct order. The SDT Master Library was modified to achieve this effect. Fortunately the Master Library provides 'hooks' that permit the re-scheduling of signals. Although the Master Library is reasonably well documented, the change proved to be an intricate task requiring analysis of the code generated by SDT. ANISEED can supply its own scheduling functions for discrete event simulation. An SDL system is simulated or validated by linking the new library with the code produced for the system. The system can then be simulated or validated as normal, whether from the command line or via the Graphical User Interface of SDT.

Discrete event simulation introduces a number of complications. Inputs with the same time-stamp (even for different processes) have to be scheduled at the same time, thus avoiding one process starving others of input. Careful investigation was also required to avoid execution loops due to the queue re-ordering strategy. SDT holds timer signals in a separate queue so they can be given priority over normal signals. ANISEED therefore needs to schedule this queue as well as the normal input queues. SDT treats continuous signals as special signals in the input queue. As usual, these are given lower priority over normal input signals *for the same process*. However for a discrete event simulation to work properly, a continuous signal in a process without normal inputs has to be scheduled before normal signals in *other* processes.

4 Abstract Sequencing Constraints

Constraints at the highest level may be given without regard to functionality. This defines gross sequencing relationships among the inputs and outputs of a component. The constraints may be given in untimed form, but are most useful when used to express timing restrictions. It is valuable to check for timing inconsistencies before any more detailed functional design is undertaken. For example, a component may not be able to produce its output in time for another one. Abstract timing constraints are particularly helpful in asynchronous design, since the clock pulses of a synchronous design are not available to coordinate actions. Abstract sequencing constraints appear in the library as 'components' of various forms. Once high-level sequencing properties have been validated, these 'components' are replaced by real ones. The following examples of sequencing constraints are drawn from the ANISEED library; the constraints exist in untimed and timed forms. For brevity the corresponding SDL is not given here.

An *N-Of* constraint requires an input event to occur *N* times before output occurs. As an example without a timing constraint, a divide-by-4 counter produces one output pulse for every four input pulses. A period during which counting occurs may optionally be given.

A *One-Of* constraint accepts just one input before producing output. For example, a bus arbiter must service just one client request during a bus cycle of some period. A second input within this period is retained until the next cycle. A variant of this constraint discards additional inputs before the period has elapsed.

An *All-Of* constraint requires all inputs of a component to be received before output is produced. For example, the inputs to an adder must be received before its output can be calculated. The order in which inputs arrive is unimportant, but all inputs may be required in some period. Unless all the inputs arrive in time, the whole constraint is re-enforced.

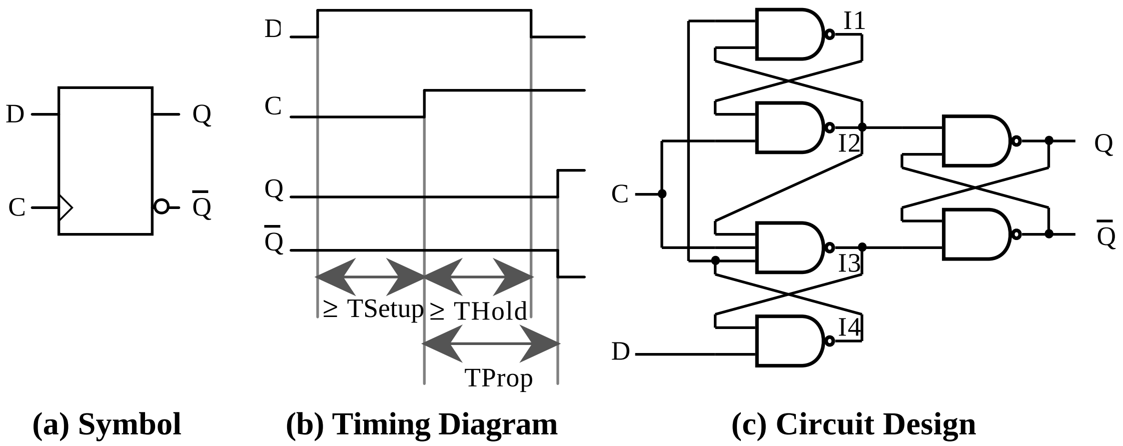
Using the same principles as the sequencing constraints in the library, the specifier may also define arbitrary constraints for complex components such as sequencers, bus controllers and interface adaptors. Sequencing constraints deal only with high-level aspects, and so are considerably simpler than the full functionality of a component.

5 A More Complex Component: A Delay Flip-Flop

5.1 Introduction

As an example of hierarchical specification, a DFF (Delay or D Flip-Flop) is the basic storage element in many hardware designs. For brevity, the detailed SDL specifications are not shown here but appear in [1]. The ANISEED library includes other kinds of flip-flops (and their simpler relatives, latches). The conventional symbol for a delay flip-flop is shown in figure 2 (a). The flip-flop stores one bit from the data input *D* under control of a clock signal *C*. The variant to be described here is positive edge-triggered, which means that the data input is stored when the clock goes from 0 to 1. After the data has been clocked in, it appears at the output *Q* after some propagation delay that depends on the hardware family. The logical complement of the output, *QBar* ($\overline{Q}$), is also available. The output value is preserved even if the *D* input subsequently changes (i.e. the flip-flop stores its input). A new data value is read only on the next rising edge of the clock signal.

Apart from the obvious propagation delay *TProp*, a D flip-flop also imposes two other timing constraints. It is required that the data input be steady for a period *TSetup* before it can



(a) Symbol **(b) Timing Diagram** **(c) Circuit Design**

Figure 2: D Flip-Flop

be clocked in. Immediately after a clock trigger, the data input must remain steady for a period *THold*. These conditions ensure that the flip-flop can reliably read in data. In particular, the flip-flop cannot be expected to deal with very short pulses of input data. The timing constraints are shown graphically in figure 2 (b).

5.2 Abstract Specification

The timing rules of figure 2 can be readily transcribed into SDL. Once the flip-flop has committed to output (i.e. after setup and hold periods), a separate process instance must be created to produce output after the propagation delay. During this period, it is necessary to allow for further inputs in parallel with the output delay. The flip-flop therefore consists of a single input process instance plus output process instances as required.

5.3 Behavioural Specification

In hardware description terminology, a behavioural specification treats the circuit or component as a black box. Only the externally visible behaviour is specified. Since the D flip-flop has very little functionality, timing considerations dominate its specification. The behavioural specification of the flip-flop thus differs little from the abstract one. In general this is not true: a processor is an example whose functional specification is very much more complex than its sequencing constraints. Apart from small changes to introduce variables for input and output values, the main addition for the flip-flop behavioural specification is how to calculate the new output value. This is generated on a rising clock edge if the input data value differs from the current output value. The flip-flop then commits to output the new value and its complement after the hold and propagation delays have expired.

5.4 Structural Specification

A structural specification concerns the internal design of a component. Structural specifications may form a hierarchy of designs at progressive levels of detail. Design stops at the level of

N1 : Nand2T < CMOSDel1, CMOSDel0, I4ˋI2ˋ, I1>
N2 : Nand2T < CMOSDel1, CMOSDel0, I1, Cˋ, I2>
N3 : Nand3T < CMOSDel1, CMOSDel0, I2ˋ, Cˋ, I4ˋ, I3>
N4 : Nand2T < CMOSDel1, CMOSDel0, I3ˋ, D, I4>
N5 : Nand2T < CMOSDel1, CMOSDel0, I2ˋ, I6ˋ, I5>
N6 : Nand2T < CMOSDel1, CMOSDel0, I5ˋ, I3ˋ, I6>

J1 : Junction2T <C, Cˋ, Cˋ>
J2 : Junction3T <I2, I2ˋ, I2ˋ, I2ˋ>
J3 : Junction2T <I3, I3ˋ,I3ˋ>
J4: Junction2T <I4, I4ˋ, I4ˋ>
J5 : Junction2T <I5, I5ˋ, Q>
J6 : Junction2T <I6, I6ˋ, QBar>

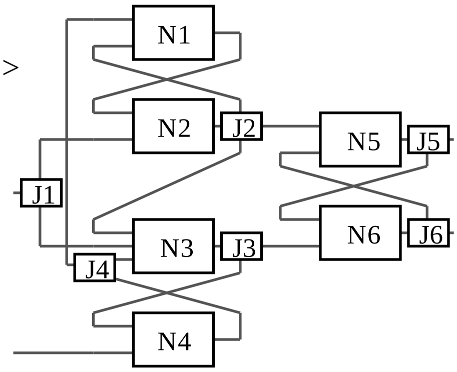


Figure 3: SDL Specification of D Flip-Flop Design

off-the-shelf components. Depending on the logic family, the off-the-shelf components may be high-level such as memories and bus controllers or low-level such as logic gates. ANISEED does not attempt to model design down to the transistor level. At each level of the design hierarchy, ANISEED may be used to specify both timing and functionality.

As an example, a typical design for a D flip-flop is shown in figure 2 (c). The internal signals *I1* to *I4* are shown. This circuit uses a number of *nand* gates (the D-shaped symbols); other flip-flop designs are possible. The *nand* gates are available from the ANISEED library as described in section 2.4.

Each component from the library is instantiated much as in section 2.5. It is convenient to use GR when specifying circuits since the structure of the electronic design and the structure of the SDL specification are very close. Basically each circuit symbol corresponds to an instance of a block type. The wires are represented as no-delay channels joining the blocks just as in the circuit diagram. The SDL equivalent of the flip-flop design is shown in figure 3; compare this with the circuit diagram in figure 2 (c). Since SDL allows channel details to be inferred from block inputs and outputs, these do not strictly need to be drawn and hence are shown as gray in figure 3. For convenience the instantiations of each block type are listed separately in the figure. Recall from section 2.4 that the context parameters of a *nand* gate are: *delays*, *inputs*, *output*. For a junction the parameters are: *input*, *outputs*. Primed signals refer to the outputs of a junction (e.g. output I' would correspond to input *I*).

5.5 Timing Analysis

Like each of the ANISEED library components, the D flip-flop was simulated and validated using the SDT toolset. Using the validator it was shown that the different levels of abstraction for the D flip-flop are equivalent in the sense that they respect the same traces. Exhaustive validation of the gate-level specification takes about one minute and 412 states. Random-walk validation of the gate-level specification takes about two minutes to explore over 211,000 states. More states are covered during random-walk validation because the state space is repeatedly explored to a fixed depth from the starting point. However random walks are not guaranteed to cover the entire state space, unlike exhaustive validation. Both timing and functionality at different levels of abstraction were shown to be consistent for the D flip-flop. Of course this is not surprising since figure 2 (c) shows a well-known design for a D flip-flop.

6 A Simple Circuit: The Single Pulser

The Single Pulser is a standard hardware verification benchmark circuit [23]. It is a clocked device with a one-bit input and a one-bit output. The purpose of the circuit is to debounce a push-button. The circuit must sense the depression of the button and assert an output signal for one clock pulse. The system should not allow additional assertions of the output until after the operator has released the button. Figure 4 shows the circuit design given in the benchmark document. It is simple and can be modelled directly using the ANISEED library. The SDL specification and its detailed analysis are given in [1], and so are omitted here. The SDL specification structure closely resembles the circuit diagram.

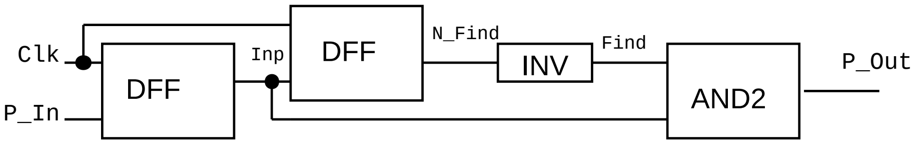


Figure 4: Implementation of Single Pulser

The circuit design was interactively simulated and automatically checked. An example of the circuit behaviour appears in figure 5; this timing diagram was generated automatically from a validator trace (MSC). The time base corresponds to a clock rate of 10 MHz (i.e. 100 ns per clock cycle). Although the functionality is correct, it was found that there is a flaw in the supposedly proven benchmark circuit! The first output pulse after power-up is longer than expected (110 ns, from 67 ns to 177 ns) instead of lasting one clock cycle. The reason for this is that on the first pulse, the second flip-flop does not need to complete its setup time (10 ns in this example). This causes the first output pulse to appear 10 ns early. On subsequent pulses the second flip-flop has to allow its setup delay to pass, so the output pulse length is correct at 100 ns (e.g. from 277 ns to 377 ns).

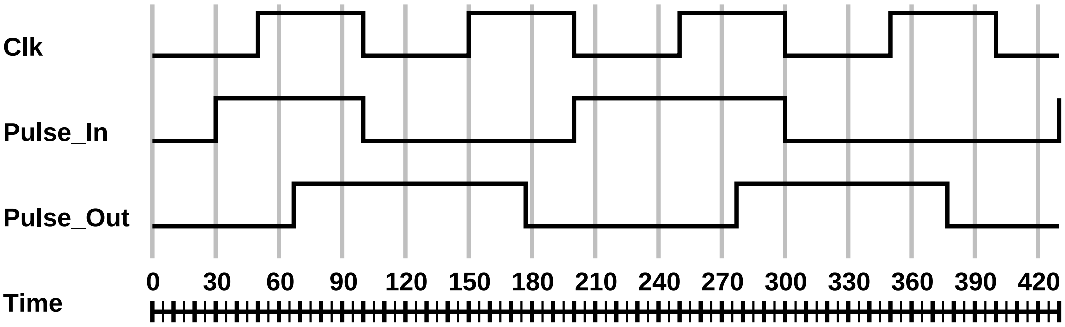


Figure 5: Timing Behaviour of Single Pulser

7 A More Complex Circuit: A Bus Arbiter

The Bus Arbiter is another standard hardware verification benchmark circuit [23]. It is a good example of a control-dominant circuit. The arbiter also presents a scalable design, which is a useful way of evaluating verification tools. The number of the arbiter cells can be chosen according to the ability of the verification tools. The purpose of the Bus Arbiter is to grant access on each clock cycle to a single client among a number of clients requesting use of a bus. The inputs to the arbiter are a set of request signals from each client. The outputs are a set of acknowledge signals, indicating which client is granted access during a clock cycle.

As shown in the structural specification of figure 6, each cell of the arbiter is moderately complex. The whole circuit consists of a number of such cells connected cyclically, e.g. the three shown in figure 7.

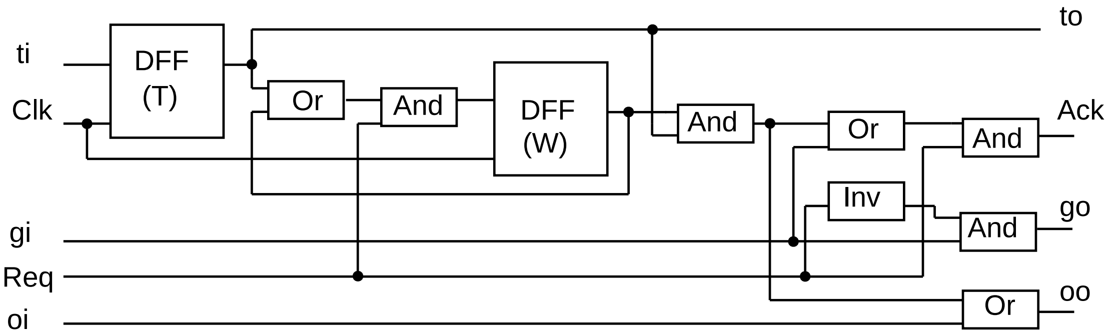


Figure 6: Design of A Bus Arbiter Cell

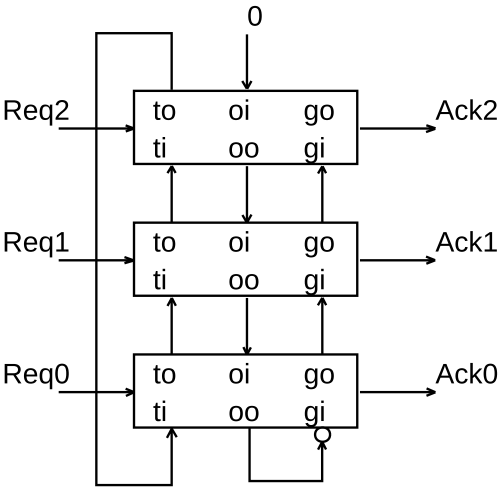


Figure 7: Interconnection of Multiple Bus Arbiter Cells

The design of the circuit will be explained only briefly here. The *ti* (token in) and *to* (token out) signals are for circulation of the token. The *to* output of the last cell is connected to the *ti* input of the first cell to form a token ring. The *gi* (grant in) and *go* (grant out) signals are related to priority. The grant of cell *i* is passed to cell *i+1*, indicating that no client of index less than or equal to *i* is requesting. Hence a cell may assert its acknowledge output if its grant

Signal	**Cycle1**	**Cycle2**	**Cycle3**	**Cycle4**
Req0	1	1	1	0
Req1	0	0	0	1
Req2	0	0	0	0
Ack0	1	1	1	
Ack1	0	0	0	1 (behavioural) 0 (structural)
Ack2	0	0	0	

Figure 8: Example of Inconsistency between Bus Arbiter Specifications

input is asserted. The *oi* (override in) and *oo* (override out) signals are used to override the priority. When the token is in a persistently requesting cell, its corresponding client will get access to the bus; the *oo* signal of the cell is set to 1. This signal propagates down to the first cell (numbered 0) and resets its grant signal through an inverter. As a consequence the *gi* signal of every cell is reset, in other words the priority has no effect during this clock cycle. Within each cell, register *T* stores 1 when the token is present, and register *W* (waiting) is set to 1 when there is a persistent request. Initially the token is assumed to be in the first cell.

This circuit is relatively challenging. In the detailed design, the SDL specification contains 56 components (over 60 concurrent processes) and 93 signals. Nonetheless, the structure of the SDL specification closely resembles the circuit diagram, so translation to SDL is straightforward. All the components are drawn from the ANISEED library.

A behavioural specification of the intended behaviour was also written, so that it might be compared to the structural specification. The behavioural specification reflects the arbitration algorithm of the circuit. Validation of the supposedly proven benchmark circuit uncovered a problem. As an example, figure 8 shows client 0 requesting the bus in the first three clock cycles. In the fourth cycle, client 0 cancels its request but client 1 begins to request access. At this point the two levels of specifications are different: the structural specification offers 0 for *Ack1*, whereas the behavioural specification offers 1 for *Ack1*.

After interactive simulation of this case, it was discovered that the circuit of figure 6 provided in the benchmark does not properly reset the *oo* (override out) signal in the following situation. In the previous clock cycle, the *W* (waiting) register of a cell is set. But in the current clock cycle, its client cancels the request and the token happens to move into the cell. In this situation, because the client has already cancelled its request it should be possible for another client to get the bus. However, the design still sets the *oo* signal to override the priority as if this client were still requesting. This means that no other client has the opportunity to access the bus in this clock cycle. Fixing the problem was much easier than finding it. The correction was to connect the *Req* signal to the *And* gate that follows the *W* register. The output of the *And* gate guarantees that the *oo* signal is always correctly set or reset according to the request signal in the current clock cycle.

A further problem was then discovered during automated validation using the random-walk approach. This achieves only 99.2% coverage despite increasing the search depth and the number of search repetitions. Analysis with the interactive timing simulator showed that this is due to the arbiter misbehaving when three clients simultaneously request access. In such a case the arbiter design grants requests to two of the clients concurrently! However the circuit behaves correctly with zero, one or two simultaneous client requests. The problem arises from a timing fault in the given design (not respecting flip-flop setup times).

8 Conclusions

It has been seen how ANISEED can successfully model digital hardware as a collection of interacting parallel components. The emphasis in ANISEED is on timing specification and analysis. This complements the work of others on hardware description and synthesis using SDL. The paper has explained the approach to modelling signals, wires, components and circuits. A library of typical components is automatically generated by the *m4* macro processor from specification templates. It was explained how abstract, behavioural and structural specifications can be given – particularly for timing constraints.

The approach has been illustrated with a variety of sample components and circuits. It is good that ANISEED can cope with standard hardware verification benchmarks. The authors were gratified to find that the approach discovered genuine problems with the Single Pulser and the Bus Arbiter – standard circuits that might have been supposed to be well verified. When the authors reported these problems, the benchmark circuit maintainers considered them to be timing rather than verification issues. This might explain why others who have verified the benchmarks have not reported these problems. Since any specification makes decisions about modelling and level of abstraction, it is also possible that others did not discover these problems due to their different approaches. (Equally, the approach of the authors might fail to identify deficiencies found by other methods.)

Work is continuing on the ANISEED library and tools. A GUI editor will be written to produce SDL hardware descriptions more directly from circuit diagrams (though the translation is relatively easy). Further case studies are being undertaken from hardware verification benchmarks. These will allow the ANISEED approach to be compared fully with those of other hardware description languages. Most SDL users concentrate on validation. Work at Stirling is also developing SDL verification techniques for timing characteristics of hardware.

Acknowledgements

Financial support from NATO under grant HTECH.CRG974581 is gratefully acknowledged. This has permitted collaboration with Dr. G. Adamis, Dr. Gy. Csopaki (who contributed to section 4 of this paper) and Mr. T. Kasza of the Technical University of Budapest. F. J. Argul-Marin thanks the Faculty of Management, University of Stirling, for supporting his work. Mrs. Ji He, University of Stirling, discovered and analysed the first arbiter design problem mentioned in section 7.

References

[1] F. J. Argul Marin and K. J. Turner. Extending hardware description in SDL. Technical Report CSM-155, Department of Computing Science and Mathematics, University of Stirling, UK, Feb. 2000.

[2] I. S. Bonatti and R. J. O. Figueiredo. An algorithm for the translation of SDL into synthesizable VHDL. *Current Issues In Electronic Modeling*, 3, Aug. 1995.

[3] E. Bounimova, V. Levin, O. Başbuğoğlu, and K. İnan. A verification engine for SDL specification of communication protocols. In S. Bilgen, M. U. Çağlayan, and C. Ersoy, editors, *Proc. 1st. Symposium on Computer Networks*, pages 16–25, Istanbul, Turkey, 1996.

[4] CCITT. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 1996.

[5] G. Csopaki and K. J. Turner. Modelling digital logic in SDL. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Proc. Formal Description Techniques X/Protocol Specification, Testing and Verification XVII*, pages 367–382. Chapman-Hall, London, UK, Nov. 1997.

[6] J.-M. Daveau, G. F. Marchioro, T. Ben Ismail, and A. A. Jerraya. COSMOS: An SDL based hardware/software codesign environment. *Current Issues In Electronic Modeling*, 8:59–88, 1997.

[7] J.-M. Daveau, G. F. Marchioro, C. A. Valderrama, and A. A. Jerraya. VHDL generation from SDL specifications. In C. Delgado-Kloos and E. Cerny, editors, *Proc. Computer Hardware Description Languages and their Applications XIII*, pages 20–25. Chapman-Hall, London, UK, Apr. 1997.

[8] C. Delgado-Kloos and E. Cerny, editors. *Proc. Computer Hardware Description Languages and their Applications XIII*. Kluwer Academic Publishers, London, UK, Apr. 1997.

[9] W. Glunz, T. Kruse, T. Rössel, and D. Monjau. Integrating SDL and VHDL for system-level hardware design. In *Proc. Computer Hardware Description Languages XI*, pages 187–204. North-Holland, Amsterdam, Netherlands, Apr. 1993.

[10] W. Glunz, T. Rössel, and T. Kruse. Hardware/software co-design using SDL. In *Proc. 2nd. International Workshop on Hardware/Software Codesign*, pages 5–21, Innsbruck, Austria, May 1993.

[11] T. Hadlich and T. Szczepanski. The ODE system – An SDL based approach to hardware-software codesign. In C. Müller-Schlör, F. Geerinckx, B. Stanford-Smith, and R. van Riet, editors, *Embedded Microprocessor Systems*, pages 269–281. IOS Press, Amsterdam, Netherlands, 1996.

[12] G. J. Holzmann. Practical methods for the formal validation of SDL. *Computer Communications*, 15(2):129–134, 1992.

[13] IEEE. *VHSIC Hardware Design Language*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1993.

[14] IEEE. *IEEE Standard Hardware Design Language based on the Verilog Hardware Description Language*. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.

[15] ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 1996.

[16] ITU. *Specification and Description Language – Common Interchange Format*. ITU-T Z.106. International Telecommunications Union, Geneva, Switzerland, 1996.

[17] A. A. Jerraya, M. Romdhani, C. A. Valderrama, P. Le Marrec, F. Hessel, G. F. Marchioro, and J.-M. Daveau. Languages for system-level specification and design. In W. Wolf and J. Staunstrup, editors, *Hardware/Software Co-Design: Principles and Practice*, pages 235–262. Kluwer Academic Publishers, London, UK, 1997.

[18] V. Levin, E. Bounimova, O. Başbuğoğlu, and K. İnan. A verifiable software/hardware co-design using SDL and COSPAN. In Z. Brezocnik and T. Kapus, editors, *Proc. COST 247 International Workshop on Applied Formal Methods*, pages 6–16, Slovenia, June 1996. University of Maribor.

[19] B. Lutter, W. Glunz, and F. J. Rammig. Using VHDL for the simulation of SDL specifications. In *Proc. European Design Automation Conference 92*, pages 630–635, New York, USA, 1992. Institution of Electrical and Electronic Engineers Press.

[20] O. Pulkkinen. SDL-VHDL integration. In K. Kronlöf, editor, *Method Integration: Concepts and Case Studies*, pages 271–307. John Wiley and Sons, Chichester, UK, 1993.

[21] M. Romdhani, A. Jeffroy, P. De Chazelles, and A. A. Jerraya. Composing Activity-Charts/StateCharts, SDL and SAO specifications for co-design in avionics. In *Proc. European*

Design Automation Conference 95. Institution of Electrical and Electronic Engineers Press, New York, USA, Sept. 1995.

[22] R. Seindal. GNU *m4* (version 1.4). Technical report, Free Software Foundation, 1997.

[23] J. Staunstrup and T. Kropf. IFIP WG10.5 benchmark circuits. http://goethe.ira.uka.de/hvg/benchmarks.html, July 1996.

[24] Telelogic. *TAU 3.5 Manuals.* Telelogic, Malmø, Sweden, June 1999.