Department of Computing Science and Mathematics
University of Stirling

Extending Hardware Description in SDL

F. Javier Argul-Marin and Kenneth J. Turner

Technical Report CSM-155

ISSN 1460-9673

February 2000

Department of Computing Science and Mathematics
University of Stirling

Extending Hardware Description in SDL

F. Javier Argul-Marin and Kenneth J. Turner

Department of Computing Science and Mathematics
University of Stirling
Stirling FK9 4LA, Scotland

Telephone +44-1786-467-421, Facsimile +44-1786-464-551
Email javargul@bbvnet.com, kjt@cs.stir.ac.uk

Technical Report CSM-155

ISSN 1460-9673

February 2000

Abstract

The use of SDL (Specification and Description Language) for digital hardware description and analysis is investigated in this report. It continues the work undertaken at the University of Stirling and the Technical University of Budapest on hardware description with SDL, offering a modular approach to hardware design in SDL.

Although SDL is widely used in the software and telecommunication community, it is not very popular with hardware designers. However, it has attracted the researcher's interest because it offers good system structuring features and the possibility of software-hardware co-design.

One way of supporting hardware engineers when translating a circuit schematic into a SDL specification is to have a library of ready-to-use or pre-defined digital components. These elements may then be used as building blocks to aid in the development of more complex electronic hardware.

The main goal of this report has been to extend an existing SDL logic library, in an attempt to reflect the range of components typically available to electronics designers. Using these libraries and a commercial tool for SDL the properties of a realistic circuit can now be investigated. Making use of these new elements, a practical case study has been carried out. The overall results clearly show that hardware description in SDL is an interesting alternative to other more traditional methods of hardware analysis.

Thanks to the Faculty of Management for the financial support for this project and, finally, thanks to the Department of Computing Science and Mathematics as a whole because it has been the perfect environment for this work. Financial support is gratefully acknowledged from NATO under grant HTECH.CRG 974581.

Table of Contents

List Of Figures

1 Introduction

1.1 Background and Context

During the last decade, hardware design has evolved from using tools for synthesis from Boolean equations and state diagrams, to synthesis from behavioural descriptions using HDLs (Hardware Description Languages). Nowadays, high-level synthesis tools are commercially available and widely used for design.

Although improving the performance of high level synthesis tools is an active research area, some researchers have also started looking at the problem of direct synthesis from system specification languages like SDL [1] (Specification and Description Language). The use of formal methods for hardware description and specification is a relatively new research area, although much of the experience and commercial tools for formal software design could be used for hardware too.

Digital hardware and software are not that different after all. They share things in common that could be exploited to achieve a better understanding of complex structures implemented as hardware elements or software routines. At a higher level of abstraction, system designs can be analysed, optimised and tested independently of the implementation. The use of formal methods for hardware analysis and the new approaches towards co-design can certainly weaken the barriers traditionally built between the two worlds, the hardware and software realms.

1.2 Scope and Objectives

This dissertation continues the work already undertaken at the University of Stirling and the Technical University of Budapest on hardware description with SDL. Kenneth J. Turner, Gyula Csopaki and Stephen D. Laing have jointly developed the foundation work in the project ANISEED (Analysis In SDL Enhancing Electronic Design) an innovative attempt to offer to electronics engineers a modular approach to hardware design in SDL.

Complex circuits can be described and analysed in ANISEED making use of a library of electronic components described in SDL. In order to extend the functionality and possibilities of ANISEED, the main goal of this dissertation has been to extend the existing SDL logic library, in an attempt to reflect the range of components typically available to electronics designers. Several components have been selected from typical device families such as tri-state logic gates, flip-flops, code converters, multiplexers, etc. The behaviour of hardware functional units has been specified by block types, and all the components stored in SDL packages to be used as generic definitions. These generic elements can now be instantiated to specify the characteristics of particular components, including parameters such as names of input and output signals, timing characteristics, propagation delays, etc.

The main outcome of this project has been an extended SDL library for future use in ANISEED. Besides, to make use of these new elements a practical case study has been carried out. The overall results clearly show that the ANISEED approach to hardware description in SDL is an interesting alternative to other more traditional methods of hardware design and analysis.

1.3 Structure of the Report

This dissertation is structured as follows:

- Chapter 1: (This chapter). Introduces the background and context of the work, establishing the goals and main objectives.
- Chapter 2: Gives some notation and semantics of the basic digital components needed to understand the following work. It also describes the state of the art in the most currently used hardware description languages. The main characteristics of SDL are presented, and the ANISEED approach is briefly described.
- Chapter 3: The approach to hardware description in SDL that has been followed in this work is presented. Some simulation and validation issues are also discussed. Explanation of how a commercial tool can be used to validate SDL descriptions of digital circuits is given.

- Chapter 4: This chapter is dedicated to tri-state devices. Some particular aspects of these components are commented on, the new elements included in the library are explained, and a detailed example is given.
- Chapter 5: Code converters and multiplexers are presented in this chapter. Common aspects and SDL descriptions of coders, decoders, BCD to decimal code converters and multiplexers are discussed.
- Chapter 6: More than twenty different types of flip-flops have been included in the ANISEED library. Different kinds of flip-flops and their timing constraints are described, discussing the solutions found to deal with the inherent complexity of all these timing aspects in SDL.
- Chapter 7: This chapter describes how the new library was constructed and how to use it. Some problems with the tool and the solutions found are also discussed. A case study shows our approach to hardware analysis in SDL in action.
- Appendix A: The most commonly used SDL symbols and notation are included for reference.
- Appendix B: A list of the new ANISEED library components.

2 State of The Art

2.1 Digital Hardware Components

Digital systems are extensively used in computation and data processing, control systems, communications, measurement, etc. Because digital systems are capable of greater accuracy and reliability than analogue systems, many tasks formerly done by analogue systems are now being performed digitally.

The design of digital systems may be divided roughly into three parts; system design, logic design and circuit design [2]. System design involves breaking the overall system into subsystems and specifying the characteristics of each subsystem. Logic design involves determining how to interconnect basic logic building blocks to perform a specific function. Circuit design involves specifying the interconnection of specific components like resistors, transistors, logic gates etc. to form logic building blocks. Most contemporary circuit design is done in integrated circuit form using appropriate computer-aided design tools to lay out and interconnect the components on a chip of silicon.

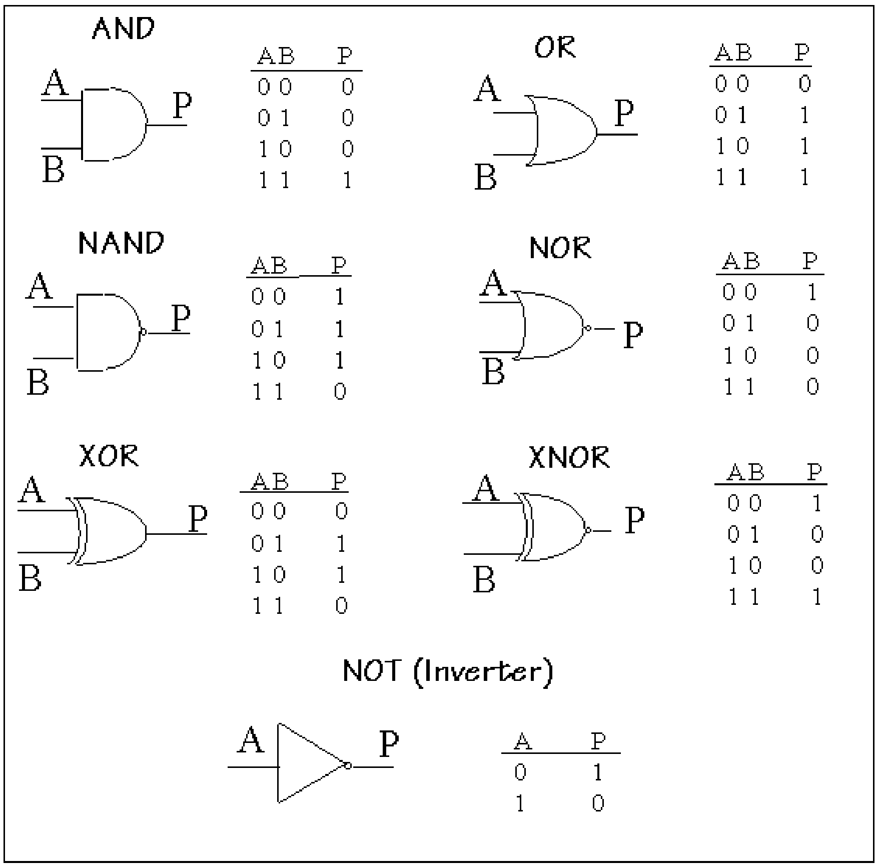
Many of the subsystems of a digital system take the form of a reactive system with one or more inputs and outputs which take discrete values. In combinational networks the output values depend only on the present value of the inputs and not on past values. However, in a sequential circuit the outputs depend on both the present and past input values. In other words, to determine the output of a sequential circuit a sequence of input values must be specified. Sequential circuits are said to have memory because they must remember something about the past sequence of inputs, while combinational networks do not.

The simplest building blocks used to construct combinational circuits are logic gates. The logic designer must determine how to interconnect these gates in order to convert the input signals into the desired output signals. The relationship between these input and output signals can be described mathematically in terms of Boolean algebra.

The basic memory elements used in the design of sequential networks are flip-flops or latches. Flip-flops can be interconnected with gates to form counters, registers and the like. The first step in designing a sequential circuit is to construct a state table or graph which describes the relationship between the input and output sequences. After doing that, there are different methods to implement sequential circuits, going from a state table or graph to a network of gates and flip-flops.

Digital logic and digital systems design are highly developed topics. The operation of logic gates and how to combine them into larger circuits and modules is well documented in the literature. Traditional digital logic design normally uses hardware components as building blocks that are available in manufacturer’s catalogues and datasheets.

The behaviour, truth tables and characteristics of the electronic components described in SDL for the ANISEED library will be gradually explained in the following chapters. As a basic reference, the symbols and truth tables for the basic logic gates (with two inputs) are presented in Figure 1.



AND

AB	P
0 0	0
0 1	0
1 0	0
1 1	1

OR

AB	P
0 0	0
0 1	1
1 0	1
1 1	1

NAND

AB	P
0 0	1
0 1	1
1 0	1
1 1	0

NOR

AB	P
0 0	1
0 1	0
1 0	0
1 1	0

XOR

AB	P
0 0	0
0 1	1
1 0	1
1 1	0

XNOR

AB	P
0 0	1
0 1	0
1 0	0
1 1	1

NOT (Inverter)

A	P
0	1
1	0

***Figure 1**. Basic logic gates symbols and their corresponding truth tables*

2.2 Hardware Description Languages

Hardware Description Languages (HDLs) are, as the name implies, languages used to design hardware. HDLs can be used to describe the functionality of hardware as well as its implementation. Nowadays, hardware description languages that resemble software-programming languages are central to digital circuit design.

Hardware description languages can describe the functionality of a piece of hardware independently of the implementation. A great advance with modern HDLs was the fact that a single language could be used to describe the function of the design and also to describe the implementation. This allows the entire design process to take place in a single language.

VHDL (Very high-speed integrated circuit Hardware Description Language) [3] and Verilog [4] are some of the most widely used HDLs nowadays. VHDL has been an IEEE Standard since 1987. It is an Ada-based language that supports the development, synthesis, and testing of hardware designs through simulation of hardware descriptions. Several synthesis, verification and simulation tools based on VHDL are commercially available. The Verilog HDL was designed and first implemented at Gateway Design Automation in 1984. Due to industry concerns about the proprietary nature of Verilog, the control of the language was eventually given to a standards committee. Verilog is now an IEEE standard that is maintained by the Design Automation Standards Committee. It is a language intended for use in all phases of the creation of electronic systems, but it is primarily used for the design of integrated circuits at various levels of abstraction.

Besides specific HDLs, some software-oriented languages have been used for hardware description too. In a paper by Janstch [5], SDL and functional languages like Erlang [6] or Haskell [7] are found appropriate to describe combined software/hardware systems.

The use of formal methods for verifying and validating complex systems behaviour is an active research area. System level specifications can be used as a basis for deriving implementations, but with a higher level of abstraction, in order to postpone implementation decisions and not to exclude any valid realization. Many intermediate refinement steps are needed to achieve a realization, gradually closing the gap between the specification and the implementation. However, several concepts supported by system level specification languages are not easily represented in hardware description languages and, sometimes, clumsy implementations are needed.

Initial work by researchers for hardware synthesis from SDL specifications was mainly exploratory. The initial objective was to demonstrate the feasibility of synthesis rather than development of practical tools or methodologies. A very common strategy has been to select a restricted synthesisable subset of SDL and then provide translators to VHDL code [8, 9,10,11]. Although SDL is widely used in the software and telecommunication community, it is not that popular with hardware designers. It has attracted interest because it offers good system structuring features, high level communication and the possibility of co-design [12,13,14].

Like SDL, LOTOS (Language Of Temporal Ordering Specification [15]) was developed for describing communications systems. The inspiration for the work reported in this report was the LOTOS-based approach to hardware description currently under development at the University of Stirling: DILL (Digital Logic in LOTOS [16, 17,18]).

2.3 SDL (Specification and Description Language)

SDL (Specification and Description Language) is an object-oriented formal specification and description language for developing the structure, behaviour and data of complex systems. SDL serves as the main international standard for protocol and system description in telecommunications, being standardized by ITU (International Telecommunication Union) in recommendation Z.100. Although SDL is widely used in the telecommunications field, it is also being applied to a diverse number of other areas.

SDL has been evolving since the first “Z.100 Recommendation” in 1980 with several updates. Object Oriented features were included in the language in 1992. For some time most tools only supported the 1988 standard and, as a consequence, a distinction was made between “SDL-88” and “SDL-92”, even though each new ITU standard has replaced the previous version. SDL-96 and SDL-2000 have offered new features, though commercial support is still to catch up.

SDL features a formal definition, i.e. rules that formally define the semantics behind each symbol and concept, and stipulates how parts of the language fit together. SDL’s formality enforces precision during specification and provides support for analysis and verification. SDL also supports dynamic features that are software oriented, like dynamic process creation and dynamic addressing. This high-level language improves productivity of the design process by letting the designer concentrate on the application problem instead of dealing with low level programming issues. The formal nature of the language also facilitates automatic generation of application code directly from SDL designs.

For systems engineering SDL is normally used in combination with other languages such as the OMT/UML object model, MSC (Message Sequence Chart) or ASN.1 (Abstract Syntax Notation). The ITU Z.105 standard defines the use of SDL with ASN.1, and the Z.120 standard defines Message Sequence Charts [19]. MSC is a graphical and textual language for the description and specification of the interactions between system components. The main area of application for Message Sequence Charts is as an overview specification of the communication behaviour of real-time systems. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation. They have been widely used to validate the hardware descriptions presented in this work.

The static structure of a system is described in SDL by a hierarchy of blocks. A block can contain other blocks, resulting in a tree structure. The behaviour of the blocks is described by one or more communicating processes, which are described by extended finite state machines (a number of states, and transitions connecting these states).

Processes are connected with each other and to the boundary of the block by signalroutes. Blocks are connected by channels. A communication through signalroutes is timeless while a communication through a channel is delayed non-deterministically. Channels and signalroutes may be both uni- and bi-directional.

Each process is composed of a set of states and transitions and has an input queue where signals are buffered on arrival. The arrival of an expected signal in the input queue enables a transition. The process can then execute a set of actions such as manipulating variables, calling procedures and sending signals. The received signal determines the transition to be executed. When a signal has initiated a transition it is removed from the input queue. Synchronization between processes is achieved using an exchange of signals.

Each process has a unique address (process identifier) which identifies it. A signal always carries the address of the sending and the receiving processes in addition to possible values. The destination address may be used if the destination process cannot be determined statically. The address of the sending process may be used to reply to a signal.

In my view, SDL is a user-friendly language, mainly due to its graphical representation, SDL/GR, in which graphical syntax is complemented by textual syntax when needed. There is also a textual phrase representation, SDL/PR, using only textual syntax. SDL/GR and SDL/PR have a common subset of textual syntax, and thus overlap each other. All new hardware elements included in this report have been developed in SDL/GR but converted into SDL/PR to be included as part of the ANISEED library. Appendix A contains the graphic representations, names and meanings of the most commonly used SDL symbols and notation.

2.4 ANISEED

The ANISEED (Analysis In SDL Enhancing Electronic Design) project has been briefly presented in chapter 1. Now it is time for a more detailed description, as it is the context in which the present work is embodied.

Initial work on using SDL for hardware description in ANISEED has been carried out at the University of Stirling (Department of Computing Science and Mathematics) and the Technical University of Budapest (Department of Telecommunications and Telematics). A paper [20] by Gyula Csopaki and Kenneth J. Turner addressed the specification and validation of digital components and digital systems using SDL in ANISEED.

Hardware engineering usually deals with relatively low-level issues and, maybe for that reason, specification and design are rather close. In software engineering a 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. As well as being the project name, ANISEED also refers to the hardware description method and the special-purpose tools and library developed within the project.

ANISEED supports the hardware engineer when translating a circuit schematic into a SDL specification, since it contains a variety of pre-defined components. Libraries in the form of SDL packages supply ready-made circuit elements and design structures. These present solutions in a form that is familiar to electronics engineers. Translation into ANISEED allows properties of a circuit to be investigated, making use of the resources available in a commercial tool for SDL [21]. Since SDL is widely used in industry and well supported by commercial tools, it is hoped that the approach will be attractive to hardware designers. Only a basic knowledge of SDL is required in order to describe and analyse circuits.

The behaviour of a functional unit is given in ANISEED by an SDL description. Block types are used to represent generic components, actual components being instances of these. Component descriptions are stored in a library as SDL named packages. When the generic definition of a component is instantiated, its parameters are set to describe the characteristics of the particular instance. Parameters usually include the names of input and output signals and timing characteristics such as propagation delays.

ANISEED follows a modular approach to hardware description. Once the design of a module is proved correct, it may be used as a building brick in higher level designs. That is, it may be treated as a black box whose internal structure is unimportant at a higher level of abstraction. A circuit design usually employs a certain number of components. Processes are therefore combined into a SDL block structure. As a block type, a structure can also be stored in an SDL package for future use.

ANISEED makes it possible to describe mixed hardware-software systems within the same framework. If the designer wishes to specify functional behaviour at an abstract level, it is usually irrelevant whether the realisation is in hardware or software. The designer merely has to specify the interfaces of the functional unit, including input and output data (structures) and timing constraints. At this level of abstraction, a functional unit can be a hardware or software element, as both realisations may be available.

The ANISEED method can also be used for specifying and analysing timing characteristics of hardware designs. The original developers [22] have concentrated on timing aspects of hardware specification and analysis, the main goal being to allow timing constraints on circuits and components to be specified and analysed at various levels. Timing may be specified in ANISEED at an abstract (overall sequencing constraints), behavioural (black-box viewpoint) and structural (internal design) level. For timing analysis, ANISEED achieves a discrete event simulation by automatically modifying the scheduling strategy of a standard SDL simulator. Another general approach, based on modified SDL descriptions, is currently being developed at the Technical University of Budapest for real-time hardware simulation in SDL.

3 Approach

3.1 General Approach to Hardware Description in SDL

Most uses of SDL for hardware description have aimed at synthesis using standard engineering tools. As has been said in chapter 2, 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 more detailed analysis.

Hardware-software co-design using SDL has also been investigated. Hardware elements are usually generated via VHDL, while software elements are generated in high level languages like C or C++. Some SDL toolsets that support co-design include COSMOS [23] and ODE [24]. A system is generally viewed as a set of communicating hardware (VHDL) and software © subsystems. The same C, VHDL descriptions can be used for both co-simulation and hardware-software co-synthesis. In ANISEED the behaviour of a functional unit is given by an SDL description. Translation to VHDL and/or C is assumed to be dealt with by other tools.

The approach followed in ANISEED deals only with discrete signals, but it models continuous signals implicitly by modelling discrete changes in them (the edges). Hardware signals are modelled as SDL signals with two parameters: the time when the signal is generated, and the logic value. The time value of an input signal records when it was generated. The time value is used to determine the time of possible output signals (according to the time delay inherent in a component). The logic value of a signal may be a single bit, but for generality a vector of bits (multi-bit) may be used. This caters for common situations such as a bus or a group of wires that is to be specified as a whole.

Time delays are often significant in the design of digital logic – especially in asynchronous circuits. It is important that the designer be able to state propagation delays and timing restrictions explicitly. Timing information appears in process parameters and in signals. The unit of time in an SDL description is at the discretion of the specifier. Integer time values are commonly used, with a typical interpretation being nanoseconds.

The wires of a circuit are normally considered to carry signals instantaneously between components. This is not strictly true, but the transmission time over a wire is usually negligible compared to the reaction time of a component. In high-speed circuits, a wire can be modelled as a delay if necessary. In digital hardware, the wires between components usually carry signals only in one direction. However bi-directional signals are possible, for example over a bus. The SDL processes representing components are connected by zero-delay channels representing the wires. As usual, channels can be uni-directional or bi-directional.

One of the problems in modelling digital logic is that the initial state of a digital system often cannot be predicted. A simple way to model initialisation is to set each output to 0. This assumption can give temporary inconsistencies when two logic gates are connected, for example two inverters in series. To deal with that, a more accurate model is used in ANISEED. Although binary signals have the value of 0 or 1, a bit variable is also permitted to have the value *X* (meaning unknown). We make X the initial state of every signal. X can be interpreted as 'unknown', 'arbitrary' or 'do not care'. This removes inconsistencies such as in the example above. The implications of having signals which can be in one of three states, 1, 0, or X, is that the SDL built-in logical operators for the Bit type, AND, NOT, OR, etc. are no longer useful. Therefore, ANISEED specifications use a library of abstract data types (ADTs) for the logical operators AndB, NandB, OrB, NorB, etc. These new operators allow for operands with value X. As an example, Figure 2 shows the truth table for the new AndB and OrB operators included in the package Bit1.

Input 1	Input 2	Output (AndB)	Output (OrB)
0	0	0	0
0	1	0	1
0	0	0	1
1	1	1	1
1	X	0	X
0	0	0	X
X	X	X	1
1	1	X	1
X	X	X	X

***Figure 2**. Truth tables for the new AndB and OrB bit operators*

Some aspects of logic design require special "components" in SDL. 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 block types ZERO and ONE that provide logic 0 and 1 respectively. It may also be necessary to specify a source of other constant values (e.g. some binary input vector). The CONSTANT block type provides a constant output given by its parameter value. Logic sources generate their constant signals at simulation time zero.

If the output of a component is not connected to anything, process output signals have to be consumed but not used. The ABSORB block type is ready to accept and absorb any signal. Note that this differs from standard hardware design: if an output of a component is unused, the engineer simply does not connect anything to it. However, the corresponding SDL process must have a route for output signals to follow (even if nothing is done with them). With a little pre-processing, this can be made invisible to the specifier. Nonetheless, it could be argued that it is desirable to force an explicit choice of what to do with each output. If an output is accidentally left unconnected, it is useful that a check of the corresponding SDL should point out the error.

In general, signals carry time and value parameters but the time parameter of a signal may be omitted when timing characteristics are not significant. This is appropriate for synchronous clocked logic, where output signals are enabled by clock pulses. In synchronous circuits, component delays can be ignored since it is assumed that the reaction time of a component is faster than the clock rate. But in an asynchronous (unclocked) circuit, exact knowledge of component delays may be necessary to avoid race conditions. Correct operation in the presence of timing constraints may be checked through simulation or through proof of correctness.

Real logic gates have a fan-out (the maximum number of other gates that can be connected to an output) and a fan-in (the maximum number of inputs). These are component limitations that can be checked by static analysis of the SDL description. Since fan-out and fan-in have an effect on the delays introduced by gates, the designer can take them into account by choosing appropriate values for the process delay parameters.

A limitation of SDL is that an output cannot be broadcast to an arbitrary number of processes. To solve this problem, ANISEED uses junction "components" that model the connecting points of wires. 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.

Making use of the solutions explained above, and exploiting the possibilities offered by commercial SDL tools, complex circuits can be described and analysed in SDL with relative ease.

3.2 Simulation / Validation approach

A standard SDL validator can be used to check for timing or functional errors in hardware design, and also for consistency between design refinements. One method widely used with software, and implemented in the SDL tool used in this work, is based on the state space exploration technique.

State space exploration emerged from research on applying formal methods to distributed, concurrent systems, and has been used for several years to analyse telecom protocols. Telelogic [25] has implemented state space exploration in its SDT Validator, which is one component of Telelogic's SDT (SDL Design Tool), the software design and development tool based on SDL that ANISEED currently uses.

Testing complex systems usually consists of two parts: conformance testing to see that the required functionality is implemented, and robustness testing to see that the system responds reasonably to unforeseen inputs [26].

Conformance testing is a complex but well-defined task, since the requirements are known when testing. Robustness testing is more difficult since it tests the unknown ways the system might run. Robustness testing becomes even more difficult for distributed systems because their concurrent nature causes interleaving of events that can be difficult to detect in advance. Traditionally, robustness testing was done manually, which is costly, tedious and prone to error. Tools like the SDT Validator automate this procedure to increase confidence that the system will work as expected. Informally, a validator executes all possible combinations of events that can happen, and reports any indication that something has gone wrong. In this way, it feeds back problems to the developer early in the process, reducing later maintenance and debugging.

Systems validation is usually based on state space exploration: the automatic generation of the reachable state space for the system. That means all possible states a system can be in, and all possible ways it can be executed. A reachability graph represents the complete behaviour of a system. The nodes of the graph represent SDL system states. The edges of the reachability graph represent SDL events that can take the SDL system from one system state to the next one. The edges define the atomic events of the SDL system. These can be SDL statements like assignments, inputs and outputs, or complete SDL transitions depending on how the state space exploration is configured. The state space of a system can be explored using different algorithms. SDT includes random walk, exhaustive exploration, bit-state exploration, interactive simulation, etc.

As its name implies, the random walk algorithm randomly traverses the state space. Each time several possible transitions are available, the validator chooses one of them and executes it. The random walk algorithm is useful as an initial attempt for robustness testing and when the state space is too large even for a partitioned bit state search.

The exhaustive exploration algorithm is a straightforward search through the reachability graph. Each system state encountered is stored in RAM. Whenever a new system state is generated, the algorithm compares it with the previously generated states to check if the state was reached already during the search. If the state was previously reached, the search continues with the successors of this state. If the new state is the same as a previously generated state in RAM, the current path is pruned, and the search backs up to try more alternatives. The exhaustive exploration algorithm requires lots of RAM, which limits its practical application. Even with a powerful machine like the one used in this work (a Sun workstation with 512 Mbytes of RAM) only very small SDL systems have been successfully validated with the exhaustive exploration algorithm. The most common result has been the system running out of memory (after several thousands of iterations), and the validation process abruptly finished.

The bit-state algorithm is fairly efficient for state space exploration. It works well, in particular if combined with a partitioned exploration strategy. This is the standard algorithm in the SDT Validator and the one I always used first to find problems and achieve 100 % symbol coverage in my specifications. Invented by Gerard J. Holzmann at Bell Laboratories in the late 1980s for large verification problems of distributed systems, the bit-state algorithm is based on using a bit array. All bits are initially set to zero to store the reachability graph. The idea is to compute a hash value, used as an index into the bit array, for each generated system state. For each newly generated system state, the algorithm computes the hash value, and checks the bit array. If the bit array has a 1 at the given index, we assume this state has been visited before, and prune the search, i.e. back up in the execution sequence and try another alternative.

During its exploration, the SDT validator checks a number of rules executed for each transition. Whenever a rule is violated, the validator saves a report that includes information about what rule was violated and the path in the state space to the violation. When the automatic exploration finishes, the reports are presented in a clickable tree overview, giving access to the system states that require investigation. The user investigates the reported situations via the validator's interactive mode. Essentially, the user gains access to the complete execution path that led to the problem, being able to walk backward and forward in this path to check the values of variables and other aspects of the system's state.

Message Sequence Charts (MSCs) can also be used to show an overview of signal interchanges between the different processes active in the investigated execution path. A "Navigator" feature allows the user to manually check alternative paths in the state space. This Navigator, combined with MSCs and watch windows to show the values of variables, are some tools that have proved very useful (but time consuming) during the validation of the systems presented in this report.

When the validator executes a transition and reaches a new system state, the situations reported may include traditional execution errors such as:

- Data operator errors (such as division by zero)
- Sub-range violation (for syntypes)
- Index out of range (for arrays)

The validator also reports problem situations specific for distributed and concurrent systems such as:

- Deadlock
- Implicit signal consumption (One process sends a signal to another process that is not able to handle it)
- Create errors (SDL allows dynamic creation of processes, so specific problems may arise)
- Output errors (Output of a signal with no receiver or too many receivers, etc).

In practice the predefined rules that the validator checks act as a fishing net that catches logical design errors. One of the most recurrent errors in my early SDL descriptions was something that, fortunately, the validator deals very well with. In systems with several timers, many different states and a certain degree of concurrency, it is very likely that some signals are not properly handled in a particular state. The validator really helps in finding bizarre combinations of signals and transitions that lead to wrong results. Even with the most careful design efforts to do things properly, it is very difficult to foresee some unpredictable (but possible) sequences of events that make things go completely wrong.

Other problems the validator detects are related to events happening at the same time in different parts of the system. For example, a signal is received from the system's environment at the same time as a timer expires, leading to two different chains of execution interfering with each other in unexpected ways.

In addition to robustness testing, SDT's validator automatically verifies consistency between message sequence charts (MSC Verification). The validator automatically verifies consistency between MSCs and the SDL system to insure that the SDL system fulfils user requirements and will solve the right problem. The verification is achieved by giving the validator an MSC as input and checking the MSC during the state space exploration. The validator matches the MSC with the possible execution sequences. When a sequence of events is found that matches the MSC, an MSC verification is reported. An MSC violation, on the other hand, is reported when the system might behave differently than the MSC prescribes. In practice, I found some difficulty in making the best use of this last feature to validate the resulting block types of the ANISEED library. It proved problematic and time consuming because of the differences in notation between my original SDL descriptions that generated the MSCs to be verified and the systems under test (instances of the new block types in the library). I had to re-arrange the names of signals in the systems under test, but new problems related to the names of parameters arose, making this feature hard to use in this particular case.

Fully validating a complex system with the SDT validator is a very time consuming task. The tool really helps in finding all possible combinations and detecting some clear error situations. However, checking that in all circumstances the simulated behaviour matches the expectations is a question of long hours and requires careful analysis by the user. The reward after a successful validation is a high degree of confidence in the quality and robustness of the description.

4 Tristate Devices

4.1 Description Issues

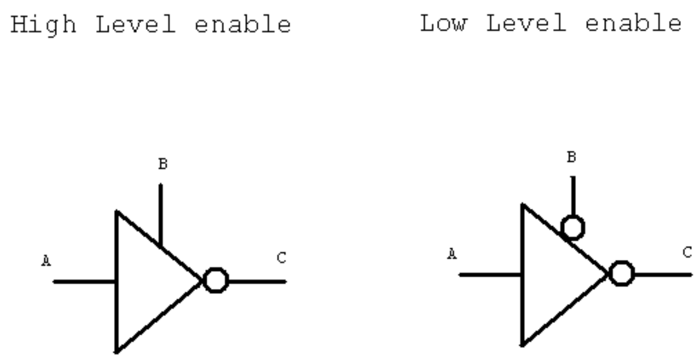
Under some adverse circumstances, a logic circuit will not operate correctly if the outputs of two or more gates are connected to each other. For example, if one gate has a “0” output (low level voltage) and another has a “1” (high level voltage), when the outputs are connected together the resulting voltage may be some intermediate value that does not clearly represent either a 0 or a 1. In some cases physical damage to the gates may result.

Use of tristate logic permits the outputs of two or more gates to be connected together, solving this problem. A tristate output is a feature of some digital electronic devices that allows a pin to either act as a normal output, driving a signal onto a line, or to be placed in third state- a high-impedance condition. This allows other outputs to drive signals onto the same line.

Tristate outputs are typically used for the connection of several digital components to a shared bus onto which any one of them may output data for the others to input. There are tristate versions of the most commonly used digital gates such as And, Or, Not, etc. Many other components such as buffers, drivers, multiplexers, latches or flip-flops are also commercially available in tristate versions.

Besides the normal signals for any electronic component, tristate devices have an additional enable input that controls the functionality and state of its outputs. Depending on the logical active level of this enable signal, two basic versions of these devices exist, low or high logical level enabled. When a tristate device is enabled (its enable input is set high or low as appropriate) it behaves like a normal component. Outputs follow the variations in inputs, and their values depend on the intrinsic behaviour implemented in the gate (And, Or, Not, Xor, etc). However, when the device is disabled, its outputs act like an open circuit. In other words, the outputs are effectively disconnected so that current can not flow. This is often referred to as a high impedance state of the output, since the circuit offers a very high resistance or impedance to the flow of current.

Figure 3 shows two different kinds of tristate inverters and their corresponding truth tables. The one on the left is a high-level enable version. When the enable input B is set to a high logic level the inverter output is enabled, and it operates normally (like any other inverter). However, when B = 0 the inverter output is effectively an open circuit. It remains in a high impedance state independently of its input value. The low level enable version (on the right) is conceptually similar, but its enable (B) input is negated, resulting in the inverter being in high impedance when B = 1.

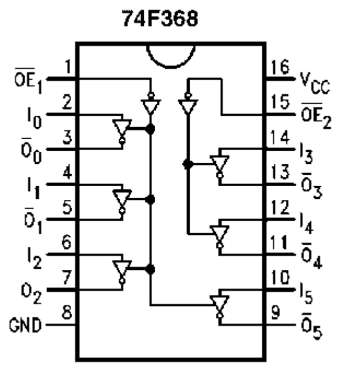


B	A	C
0	0	High Imp.
0	1	High Imp.
1	0	1
1	1	0

B	A	C
0	0	1
0	1	0
1	0	High Imp.
1	1	High Imp.

***Figure 3**. Symbols and truth tables for tristate inverters*

As an example of the tristate components available on the market, figure 4 shows the internal structure of a hex inverter buffer with tristate outputs (74F368 series). Six identical inverters are integrated in the same chip. Two different enable signals (OE1 and OE2) control the behaviour of four and two inverters respectively. As shown in the figure, these enable inputs are internally inverted and, for that reason, an output Ox is in high impedance when its corresponding enable input is at high level.



Inputs		Output
$\overline{OE}$	I	$\overline{O}$
L	L	H
L	H	L
H	X	Z

L = LOW Voltage Level X = Immaterial
H = HIGH Voltage Level Z = High Impedance

***Figure 4**. Hex inverter buffer with tristate outputs*

Timing characteristics are an intrinsic part of digital hardware behaviour. Manufacturers offer timing values for their products in datasheets where AC and DC characteristics, recommended operating conditions and absolute maximum ratings are given. Parameters such as propagation delays, set-up and recovery times, minimum pulse widths, etc. are important elements to be taken into account by digital designers. All these parameters will be dealt with and explained in the following chapters, where SDL descriptions for components with timing constraints are presented.

Tristate components are not especially complex as far as timing is concerned. They have, however, some further complexity in relation to their equivalent (non tristate) components. The existence of a new enable-disable input makes it necessary to deal with new propagation delays. Datasheets include switching characteristics for tristate inverters like the one presented in figure 4. Minimum, maximum and typical values for the propagation delays are given (in nanoseconds) as tested by the manufacturer under certain conditions. Names like TpLH and TpHL are commonly used to indicate propagation delays for Low to High and High to Low level output transitions respectively. These two values are common to any inverter (non tristate inverters also have these two parameters). They represent the time needed by the inverter to toggle its output after a change in its input. The actual values strongly depend on the technology and family of the device, but it is common to find different values for High to Low and Low to High transitions. The reason for this discrepancy is normally due to different parts of the internal circuit and even different levels of logic being involved in one or another kind of transition.

Another delay normally given for tristate components is the time that the gate needs to re-establish its output after receiving an enable input signal. This delay assumes that the output was in high impedance when the enable input was received. Finally, another delay represents the time between a disable signal being received and the output being changed to high impedance.

4.2 Library components

To construct a new SDL package for tristate components a divide-and-conquer approach was followed. It was decided to start by describing two different basic tristate gates in both high and low level enabled versions. An inverter and a two input And gate were selected as it was clear that all tristate gates, independently of the function implemented (And, Or, Nor, Nand, etc.) would have a very similar structure. After finding a solution for the ones selected as representatives, many things could be automatically applied to the others. This approach has been extensively applied to hardware description at later stages in this work. It is difficult and time consuming to find an SDL description for a new hardware component, especially if a thorough validation is performed to make sure that the description exactly matches the expected behaviour. However, after finding a valid solution it can be used to describe other hardware components with similar structure. There is no need to construct and validate SDL descriptions for, say, three-input Or and Nor gates. Timing constraints, usually the most difficult part of the specification, are identical in both cases. Only the logical function implemented by the gate, and perhaps the values for the delays (that can be selected as parameters), are different. Untimed versions of the components are even easier, as they only need to omit time parameters in signals and delays (timing characteristics are no longer significant in these devices). It was obvious that this circumstance had to be exploited, and it certainly was.

Figure 5 shows the SDL system constructed to specify a tristate positive-level enabled And gate with two inputs. This hardware element is described as a single SDL block with two data inputs, one enable-disable input and one output. Communication between the block and the environment is performed by means of the channels *Ch1* to *Ch4*. Channels *Ch1* and *Ch2* carry input signals from the environment to the gate (*SIp1* and *SIp2*). Channel *Ch4* corresponds to the enable signal *SE*. Finally, the output pin of the gate is represented by channel *Ch3* and signal *SOp*. All these signals are declared in a text box in the top-left corner of the figure. As described in chapter 3, signals carry time and value parameters. (We are dealing with timed versions of the gates, as the untimed ones are just a simplification.) "*Bit1lib*" is the SDL package with abstract data types for the bit operators that was described in chapter 3. It is referenced in the specification by means of a "*use*" clause.

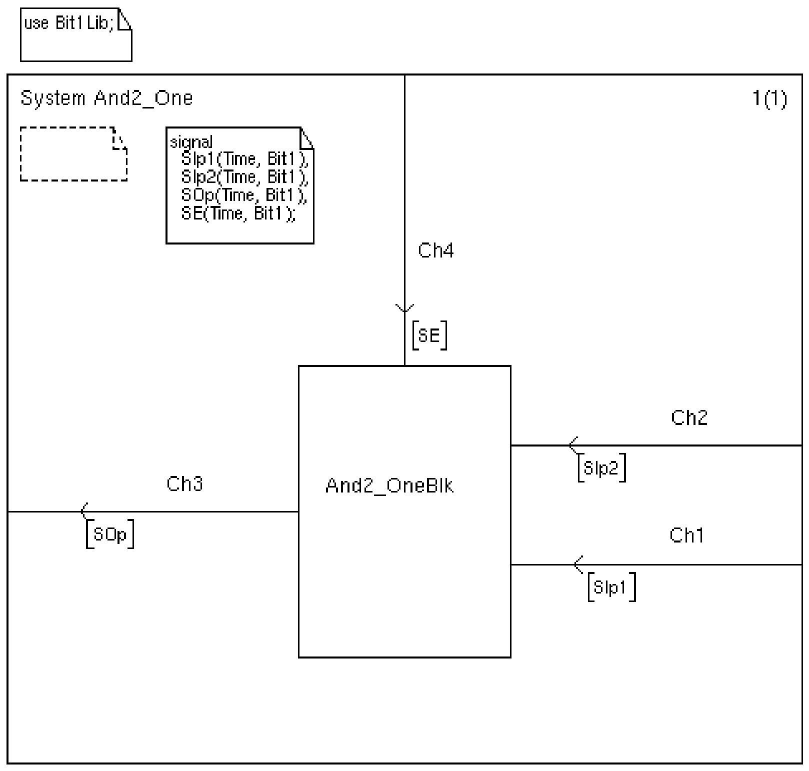


Figure 5*. SDL system diagram for a tristate AND gate with two inputs*

When signal *SE* carries a positive logical value, the gate is enabled (after the corresponding delay). When the gate is enabled, the values of *SIp1* and *SIp2* determine the output. It is calculated by applying the bit operator *AndB* to the input values. Whenever the inputs change, the output follows them accordingly, but the variation in output is not instantaneous, since propagation delays must be respected.

As noted in chapter 2, SDL follows a hierarchical structure in which systems are composed of blocks, blocks contain processes and so on. A single process could be used to describe the functionality of our tristate And gate, but after some attempts at dealing with timing constraints in the gate, a solution consisting of two different processes proved to be clearer and easier to implement. As shown in figure 6, the process named *And2_One* receives the two input signals that come from the environment and outputs signal *SOp*. Another process (*Enable*) receives signal *SE*, dealing with timing constraints in the enable-disable signal. Both processes are marked (1,1) meaning there is exactly one instance of them.

Another internal signal has been included. Notice that signals *SE*, *SIp1*, *SIp2* and *SOp* were also present in figure 5. They are external signals between the gate and the environment. However, *SEnable* is an internal signal declared within the block. It goes only between communicates processes and does not have any direct relationship with the environment. As we will see shortly, *SEnable* follows the variations in signal *SE* but only after process *Enable* has dealt with the enable-disable timing aspects. This way of dividing the specification into several processes makes the whole solution simpler. With a single process, the number of different states the system can be in grows alarmingly. Four different delays and two timers had to be considered, and things tended to become complicated even in a simple device like the one discussed here. A similar approach has been used to describe other hardware elements such as the flip-flops presented in chapter 7.

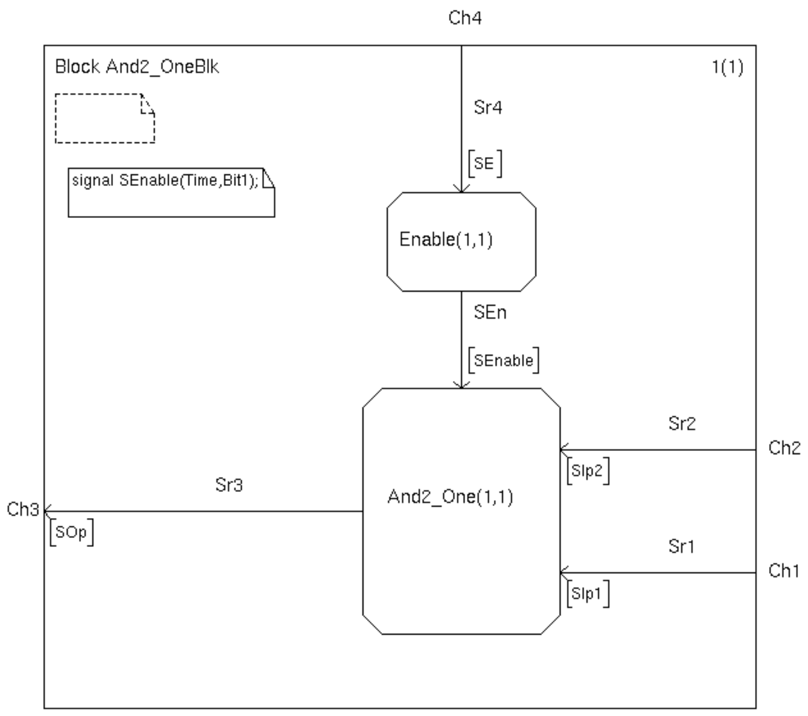


***Figure 6**. Processes contained in the block shown in figure 5*

It is out not practicable to present a detailed description of all the SDL specifications written during the project, but some SDL-GR diagrams may help to understand and illuminate the most interesting points of some components. As an example, consider the process *Enable*. As shown in figure 7, it is rather simple. As far as the enable-disable behaviour is concerned, the tristate gate can only be in one of two states, ready to receive enable-disable inputs or delaying a previous input. To avoid the temporary inconsistencies in the initial state of digital logic mentioned in chapter 3, all signal values are initialised to X (unknown or arbitrary) during start-up. A portion of SDL code deals with *BE* (the value of signal *SE*) being undetermined and randomly chooses a value of 0 or 1 for it. Signal *SEnable* is sent at time 0 with the random value chosen. When an enable signal *SE* is received, it contains the time at which it is generated, *TIp*, and its logical value *BIe*. Depending on the value, the corresponding delay is selected: *TDelayEnable* when the gate is going to be enabled and *TDelayDisable* in the other case. These two delays can be set as parameters, so the user can give the particular values for a gate in a circuit.

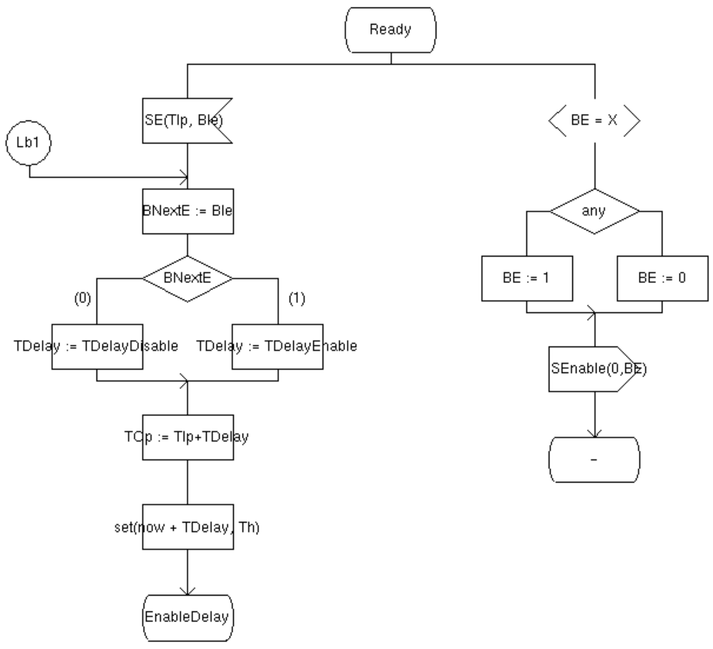


***Figure 7**. SDL-GR representation of the process Enable in its Ready state*

Once the appropriate delay has been chosen, the output time for the signal *SEnable* is calculated and a timer *Th* is set. While the timer is running the process enters a wait state. Basically two things can happen during this period (figure 8).

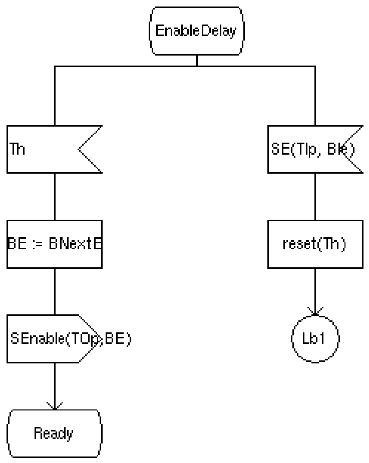


***Figure 8**. Process* Enable *waiting for the timer to expire*

Either the timer expires and then the signal *SEnable* is sent (at the calculated output time *TOp* with the received value *BE*) to the process *And2_One*, or another signal *SE* is received. In this case the delay has not been completed, in some way we could say respected. The gate did not have enough time to complete the previous transition and now it has to deal with new changes. The timer is reset and a new attempt is made to follow the inputs. Could some strange things happen then…?

Unpredictable behaviour is something really difficult to specify. Datasheets do not explicitly say what happens when timing constraints are not respected. The circuit will certainly behave in some way. Its outputs will have a certain logical value, but these may be random or hard to predict. Manufacturers only guarantee that their products behave in a certain way when the devices are operated as expected. A high level enabled tristate inverter, for example, will set its output to high impedance a certain number of nanoseconds after its enable input has gone low. If during this delay the enable input changes again, the inverter will certainly not be able to reach or maintain the high impedance state.

There is some degree of non-determinism in the behaviour of electronic hardware. This is certainly true at start-up, since every chip will set its outputs to certain values that cannot be easily predicted. We have modelled this in SDL using the value X (unknown) and giving random values to outputs at start-up. However, there is another chance for non-determinism when timing constraints are not respected. The hardware will certainly behave somehow, but how can this be specified? Well, this is something that made me think for a while and more than once made me think that I was getting everything completely wrong. It is impossible to specify something that real hardware cannot guarantee. We just can make sure that our SDL system behaves like real hardware when it is operated under the adequate conditions. As a further issue, what would happen if the inputs of a gate were continually changed at a faster rate than the propagation delays for the gate? Well, it would certainly not follow the inputs, so its output would not be the, say, And combination of its inputs at any given moment.

Coming back to our *Enable* process, we can just guarantee that if the enable-disable signal is set and the corresponding delays respected, the gate will change from normal functioning to high impedance or vice versa. If the delays are not respected the system will try its best to follow the demanding inputs, but no success can be guaranteed.

The process *And2_One* shown in figure 6 implements the behaviour of the gate without having to be bothered about timing aspects in enable-disable signals. This process has four different states: *Ready, HighImp, Waiting or HighImpWait.*

The gate is *Ready* (figure 9) when the output is already the logical And of the two inputs and it is ready to receive new inputs.

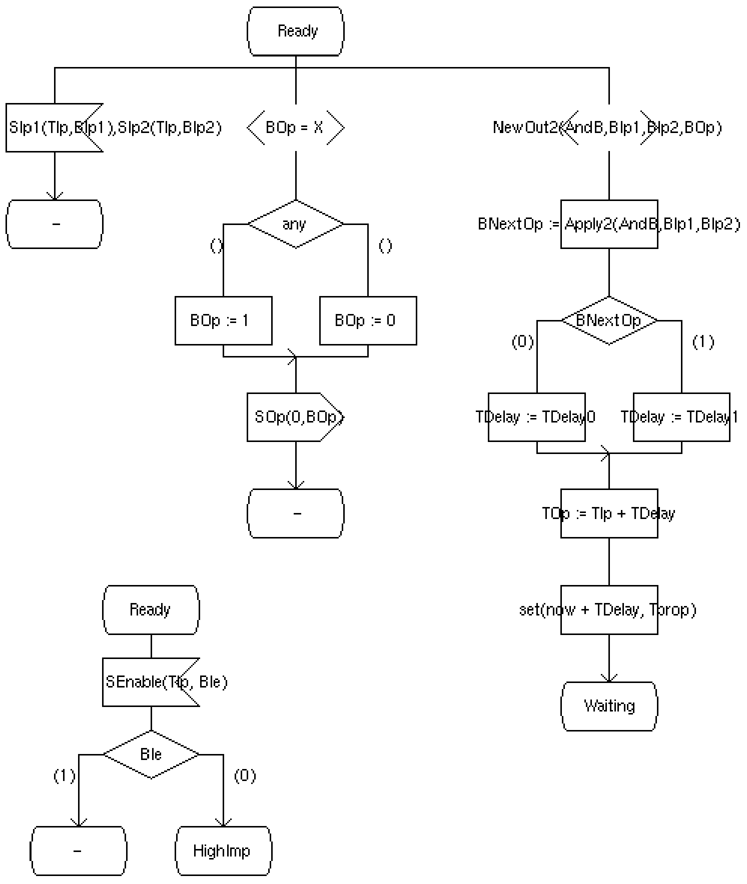


***Figure 9**. SDL description for the Tristate And2 gate in its Ready state*

As shown in the figure, a similar approach to the one used in the process *Enable* is followed here: the output value *BOp* is randomly initialised to 1 or 0 at start-up. While in the *Ready* state the process *And2_One* can receive three different signals: *SIp1*, *SIp2* and *SEnable*. New input values (*BIp1* and *BIp2*) are received at their corresponding time (*TIp*). *TIp* represents the last input time, that is, the time when the last input has been received. The bit operator *NewOut2* (included in the package *Bit1*) is used to determine if a new output is required as two inputs generate a different value. It applies the operator *AndB* to inputs and compares the resulting value with the current output, generating a Boolean (true if a new output is needed). *Apply2* is also a bit operator contained in the package *Bit1*. It is used to calculate the resulting AndB value of the two inputs. This resulting value determines whether the next transition is going to be from low to high logical level or vice versa. This is important to find the corresponding propagation delay that has to be used to set the timer. After setting the timer, the process enters the state *Waiting*.

Some problems arose with bit operator names being overloaded (the C compiler that translates C code generated from SDL complained quite a lot). To get round this problem, Ken Turner has developed a new version of the package *Bit1* with slightly different names for the operators.

While in the *Ready* state an enable-disable signal can also be received. The gate we are talking about is high level enabled, so only when *BIe* (the value of signal *SEnable*) is 0 does the gate enter the high impedance state. Notice that no timing aspects in the signal *SEnable* are considered here. These timing constraints have been dealt with in the process *Enable*. This process sends the signal *SEnable* only when the enable or disable propagation times have been completed. When the process *And2_One* receives the signal *SEnable,* it just responds to it instantaneously. If we were not using two different processes several more states would be needed, and the overall description would be far more complicated.

Waiting (figure 10) represents the state where the gate has received new inputs
that require a change in output to be made. The gate is in some way busy trying to modify its output, and to complete this task some time is needed.

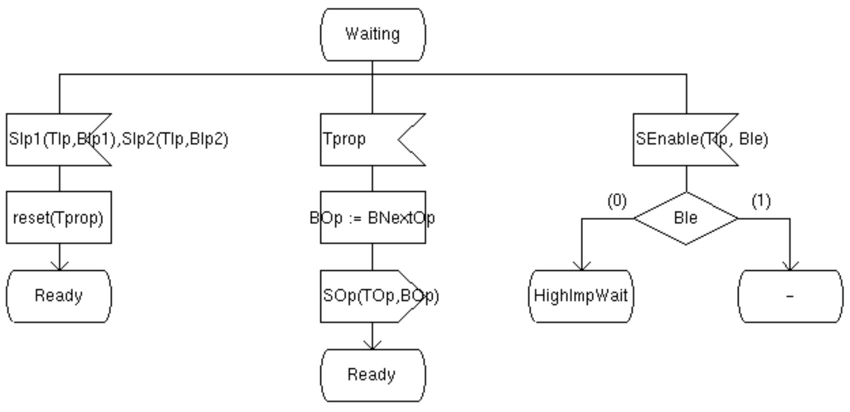


Figure 10. *Tristate gate waiting for the propagation delays to finish*

If further input signals are received while waiting, the timer is reset, and the gate goes to its *Ready* state again. If a new output needs to be generated, the gate will enter the wait state again but after setting a new timer.

If the timer expires while waiting, the output is made available with value *BOp* and time *TOp*. This output time was calculated before starting the timer.

The last option contemplates the possibility of receiving an enable-disable signal while waiting. If the value of this signal is "1" the gate goes on waiting, as it was already enabled. If the value is "0" the gate goes on waiting but now in a different state (*HighImpWait*). The gate waits for propagation delays to be finished, but the output will no longer follow the input as it is now in high impedance.

Maybe we should say something about how to best model "high impedance" in SDL. Initial thoughts were oriented towards some sort of special output value. The values used for signals so far are 1, 0 or X, this last one only used as an initial value. Adding another value such as Z to represent high impedance would not offer new features to ANISEED, and it would make bit operators far more complicated. It was then decided not to modify output values to represent high impedance. A gate in high impedance will not follow variations in its inputs, and its output will remain in the logical level that it was before entering in high impedance. The value of the output while in high impedance is not significant. To be exact, a gate in the high impedance state does not output at all. Only the fact that it no longer follows the input is modelled, and that it no longer interferes with other possible signals connected to it.

HighImpWait (figure 11) models the situation where the gate is in high impedance but the inputs have changed, or the gate was waiting and a disable signal was received. The output state will remain in high impedance as it does not depend on the

inputs, but the theoretical output value must be calculated, just in case the gate returns to its normal operating conditions after an enable signal has been received. Timers are then normally used, but when they expire the output is not changed unless the gate leaves the high impedance state.

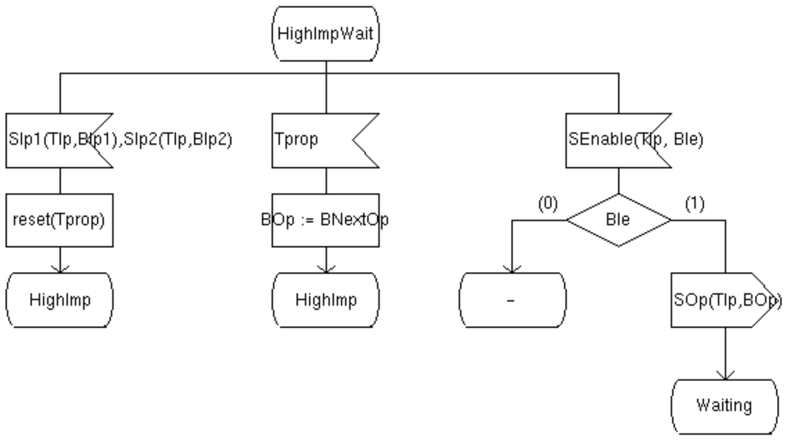


Figure 11. *Waiting while in high impedance*

HighImp (figure 12) represents the state where the gate is disabled so the output is in high impedance independently of the input values. Notice that despite being in high impedance when the inputs are received, the corresponding output value is calculated and the propagation timer is set. This is needed to re-establish the output to the right value and at the corresponding time when the gate receives an enable signal again.

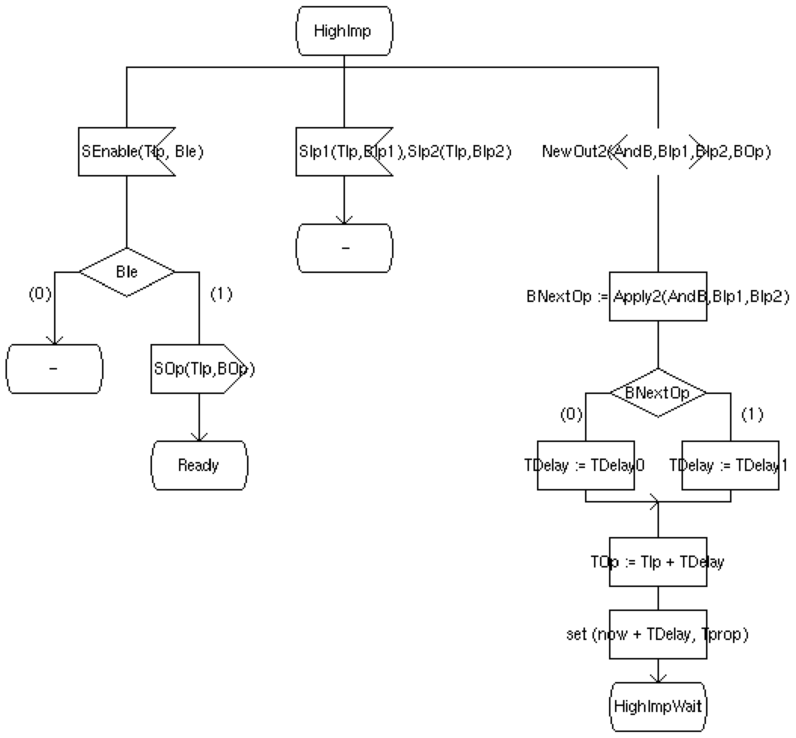


Figure 12. *Tristate gate in high impedance*

4.3 Validation

Testing and exhaustive exploration of the tristate SDL descriptions was performed with the validator included in SDT. Even with the help of the tool validation is an arduous task, maybe not for all descriptions but certainly for the more complex ones. Four SDL specifications, corresponding to high and low level enabled versions of an And2 tristate gate and an inverter were constructed and tested. 100% symbol coverage was achieved in all cases, and no error reports were given in the final versions. A bit state exploration for an And2 gate reported the following results:

The power walk algorithm was also used, and 100% symbol coverage was easily achieved:

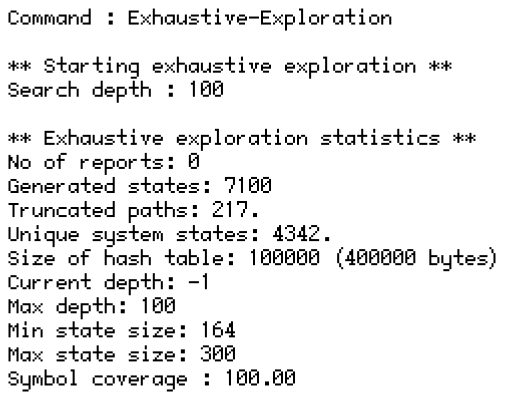
```
Command : bit-state

** Starting bit state exploration **
Search depth    : 100
Hash table size : 1000000 bytes

** Bit state exploration statistics **
No of reports: 0.
Generated states: 7097.
Truncated paths: 217.
Unique system states: 4339.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 7343
Collision risk: 0 %
Max depth: 100
Current depth: -1
Min state size: 164
Max state size: 300
Symbol coverage : 100.00
```

Even exhaustive exploration, something not very easy to achieve as will be discussed later, was feasible for the And2 tristate gate:

After examining the Message Sequence Charts resulting from the power walk algorithm, and testing the system with the navigator, a high degree of confidence in the goodness of the tristate components was achieved.



```
Command : Exhaustive-Exploration

** Starting exhaustive exploration **
Search depth : 100

** Exhaustive exploration statistics **
No of reports: 0
Generated states: 7100
Truncated paths: 217.
Unique system states: 4342.
Size of hash table: 100000 (400000 bytes)
Current depth: -1
Max depth: 100
Min state size: 164
Max state size: 300
Symbol coverage : 100.00
```

The new tristate components in the library are summarised in appendix B. A short description of how to instantiate them with the appropriate parameters is also given.

5 (De)coders and (De)multiplexers

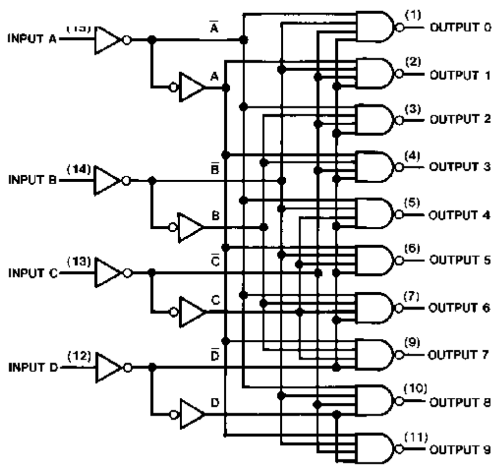
Several devices such as encoders, decoders and multiplexers are presented in this chapter. These components are commonly available as MSI (Medium Scale Integration) circuits, and can be used for different purposes. Several SDL descriptions have been created for these families of devices, and two new packages (*aniseed_coder* and *aniseed_mux*) added to ANISEED's new library.

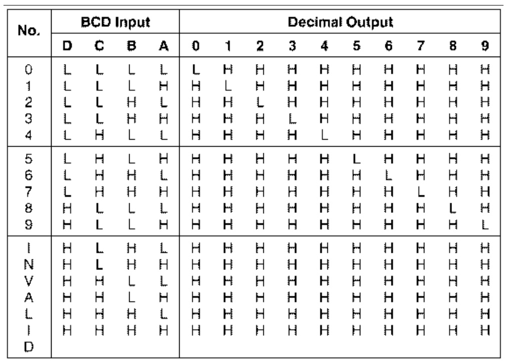
5.1 Description Issues

5.1.1 BCD-to-Decimal Decoders

In digital systems, binary representations are the most efficient way to store numbers and compute results. However, binary numbers are not easy to convert from or to decimal numbers, for human use for example. If efficiency of storage and speed of computation are not critical, Binary Coded Decimal (BCD) number representations may be preferable because they are easier to convert to a human-compatible format. BCD numbers are divided into 4-bit groups; the bits within each group are binary weighted, but may take on the values from only 0 to 9. Each group or BCD digit has a weight corresponding to a power of 10. Note that an 8-bit BCD number may represent integers from 00 to 99, while an 8-bit binary number may represent values from 0 to 255.

One possible design for a BCD-to-Decimal decoder and its corresponding truth table is shown in figure 13. It consists of eight inverters and ten, four-input Nand gates. The inverters are connected in pairs to make BCD input data available for decoding by the Nand gates. Full decoding of input logic ensures that all outputs remain off (this means high level in this example) for all invalid input conditions. There are similar versions in positive logic, where the "off" state is a low logic level.





No.	BCD Input				Decimal Output									
	D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	L	L	L	L	L	H	H	H	H	H	H	H	H	H
1	L	L	L	H	H	L	H	H	H	H	H	H	H	H
2	L	L	H	L	H	H	L	H	H	H	H	H	H	H
3	L	L	H	H	H	H	H	L	H	H	H	H	H	H
4	L	H	L	L	H	H	H	H	L	H	H	H	H	H
5	L	H	L	H	H	H	H	H	H	L	H	H	H	H
6	L	H	H	L	H	H	H	H	H	H	L	H	H	H
7	L	H	H	H	H	H	H	H	H	H	H	L	H	H
8	H	L	L	L	H	H	H	H	H	H	H	H	L	H
9	H	L	L	H	H	H	H	H	H	H	H	H	H	L
I	H	L	H	L	H	H	H	H	H	H	H	H	H	H
N	H	L	H	H	H	H	H	H	H	H	H	H	H	H
V	H	H	L	L	H	H	H	H	H	H	H	H	H	H
A	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	H	H	H	L	H	H	H	H	H	H	H	H	H	H
I	H	H	H	H	H	H	H	H	H	H	H	H	H	H
D														

***Figure 13**. Schematic and truth table for a BCD-to-Decimal decoder*

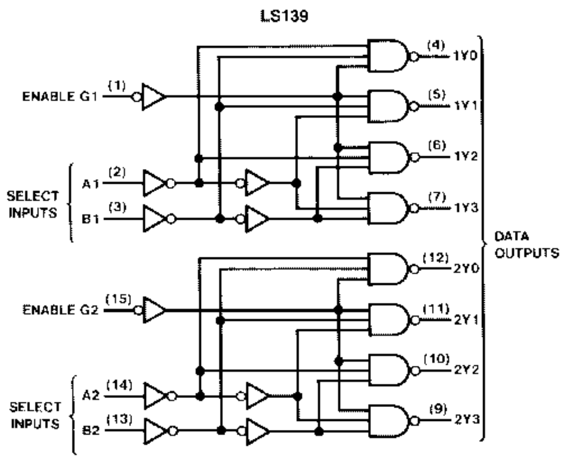
Switching characteristics for a BCD to decimal decoder usually include two different propagation delays. After a variation in inputs, *TpLH* represents the time needed to set the outputs that must be high, and *TpHL* the delay for the outputs that must be changed to a low logic level.

5.1.2 Decoders/Demultiplexers

Decoders are widely used in memory-decoding or data-routing applications. There are some versions to be used with high-speed memories that offer very short propagation delay times. The delay times of these decoders are usually less than the typical access time of the memory, and this means that the effective system delay introduced by the decoder is negligible.

A decoder such as the 74LS138 decodes one-of-eight lines, based upon the conditions at three binary select inputs. Two active-low and one active-high enable inputs reduce the need for external gates or inverters in circuits with more bits. A 24-line decoder can be implemented with no external inverters, and a 32-line decoder requires only one inverter. Exactly one of the output lines will be active (1 or 0 depending on the logic of the device) for each combination of values of the inputs.

Other devices such as the 74LS139 (figure 14) comprise two separate two-line-to-four-line decoders in a single chip. It also includes an active-low enable input that can be used as a data line, making it possible to use the device both as a decoder (while the *Enable* input is active the two *Select* inputs are decoded) or as a demultiplexer (depending on the *Select* values, the *Enable* input is sent to the desired output).



Inputs			Outputs			
Enable	Select					
G	B	A	Y0	Y1	Y2	Y3
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H = High Level, L = Low Level, X = Don't Care

***Figure 14**. Schematic and truth table for a decoder/demultiplexer 74LS139*

Typical switching characteristics for a demultiplexer usually comprise 4 propagation delays. *Select*-to-output and *Enable*-to-output reaction times are usually different. This circumstance, combined with another two propagation delays for Low-to-High and High-to-Low output transitions, lead to the 4 delays shown in figure 15.

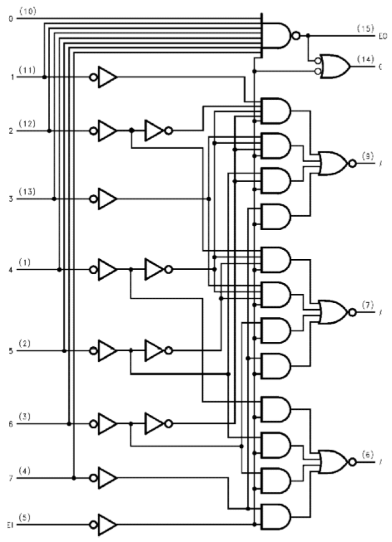
Symbol	Parameter	From (Input) To (Output)	R_L = 2 kΩ				Units
			C_L = 15 pF		C_L = 50 pF		
			Min	Max	Min	Max	
t_{PLH}	Propagation Delay Time Low to High Level Output	Select to Output		18		27	ns
t_{PHL}	Propagation Delay Time High to Low Level Output	Select to Output		27		40	ns
t_{PLH}	Propagation Delay Time Low to High Level Output	Enable to Output		18		27	ns
t_{PHL}	Propagation Delay Time High to Low Level Output	Enable to Output		24		40	ns

***Figure 15.** Switching characteristics for the device 74LS139*

5.1.3 Encoders

The terms encoder, decoder and code converter are often used interchangeably. Encoders and decoders are widely used in communications. An encoder basically converts its input into an output code with a fewer number of lines. A decoder is later used to re-construct the original representation of the data again.

Priority encoding ensures that only the highest order input data line is encoded. An 8-to-3 priority encoder accepts 8 input request lines 0–7 and outputs 3 lines A0–A2. Figure 16 shows the logic diagram and truth table for one of these devices (74HC148). All data inputs and outputs in this particular component are active at the low logic level. This device also includes cascading circuitry (enable input EI and enable output EO) to allow octal expansion without the need for external circuitry, but these inputs have not been included in the SDL specification.



Inputs								Outputs		
0	1	2	3	4	5	6	7	A2	A1	A0
X	X	X	X	X	X	X	X	H	H	H
H	H	H	H	H	H	H	H	H	H	H
X	X	X	X	X	X	X	L	L	L	L
X	X	X	X	X	X	L	H	L	L	H
X	X	X	X	X	L	H	H	L	H	L
X	X	X	X	L	H	H	H	L	H	H
X	X	X	L	H	H	H	H	H	L	L
X	X	L	H	H	H	H	H	H	L	H
X	L	H	H	H	H	H	H	H	H	L
L	H	H	H	H	H	H	H	H	H	H

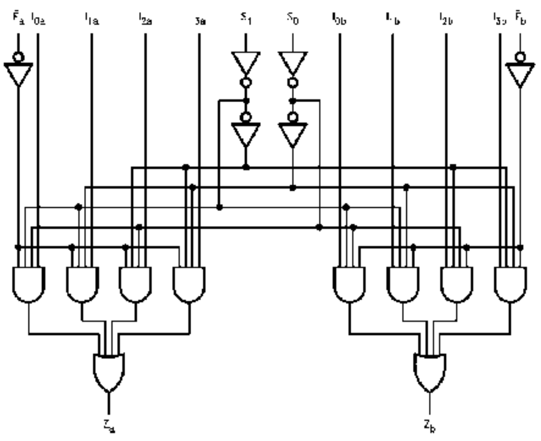
H = HIGH
L = LOW
X = Irrelevant

Figure 16. *8 to 3 encoder, logic diagram and truth table*

5.1.4 Multiplexers

A multiplexer (or data selector) has a group of data inputs and a group of control inputs. Control inputs are used to select one of the data inputs and connect it to the output terminal. Multiplexers are commonly available in integrated circuit packages in several configurations: quadruple 2-to-1, dual 4-to-1, 8-to-1 and 16-to-1. In general, a multiplexer with *n* control inputs can be used to select any one of 2^n data inputs. Multiplexers are frequently used in digital system design to select the data that is to be processed or stored. They can also be used to implement combinational logic functions. A 4-to-1 multiplexer can realize any 3 variable functions with no added logic gates.

Figure 17 shows a dual 4-input multiplexer (74F153). This device is a high-speed multiplexer with common select inputs. The two buffered outputs present data in the true (non-inverted) form. It can select two bits of data from up to four sources under the control of the Select inputs (*S0*, *S1*).



Select Inputs		Inputs (a or b)					Output
S_0	S_1	$\overline{E}$	I_0	I_1	I_2	I_3	Z
X	X	H	X	X	X	X	L
L	L	L	L	X	X	X	L
L	L	L	H	X	X	X	H
H	L	L	X	L	X	X	L
H	L	L	X	H	X	X	H
L	H	L	X	X	L	X	L
L	H	L	X	X	H	X	H
H	H	L	X	X	X	L	L
H	H	L	X	X	X	H	H

Figure 17. *Dual 4-input multiplexer, logic diagram and truth table*

A multiplexer such as this is the logic implementation of a 2-pole, 4-position switch, where the position of the switch is determined by the logic levels supplied to the two *Select* inputs. A less obvious application is to use this device as a function generator: it can generate two functions of three variables. This is useful for implementing highly irregular random logic or functions that involve a complex gating structure.

5.2 Library Components

The new SDL packages for multiplexers and coders in ANISEED's library are based on SDL descriptions of components that were considered representative, namely a 4-to-1 multiplexer, an 8-to-3 encoder, a 2-to-4 decoder, a BCD-to-Decimal decoder and a 2-to-4 demultiplexer.

There are some common aspects in all these devices that can be specified in a similar way. For example, almost all them share the same number of possible states (in the SDL sense). A BCD-to-Decimal decoder and a demultiplexer, for example, can basically be in one out of three possible states: ready for new inputs, waiting after a new input has been received but no outputs have changed yet, or waiting after some outputs have changed but not the others yet. In this last case two consecutive states are involved: waiting for low-to-high or high-to-low output transitions to occur. The relative values of the low-to-high and high-to-low propagation delays will determine which state of these two will happen first. This has been modelled in SDL making use of four states, namely *Ready*, *WaitingAll*, *WaitingHL* and *WaitingLH*.

Timing constraints in decoders and multiplexers are rather straightforward. Even when four different delays are involved (such is the case in a demultiplexer, for example), the number of possible states the device can be in does not grow exponentially. Reaction times in a demultiplexer will be different depending on which input (*Select* or *Enable*) has caused the transition. Even if response times are different (requiring a more complex specification) the device still has the same number of states: ready, waiting for all outputs to change, waiting for some outputs to go high after clearing the others or vice versa. For this reason, solutions including a single process per block (as shown in figure 18) seemed to be clear and simple enough, so they have been used.

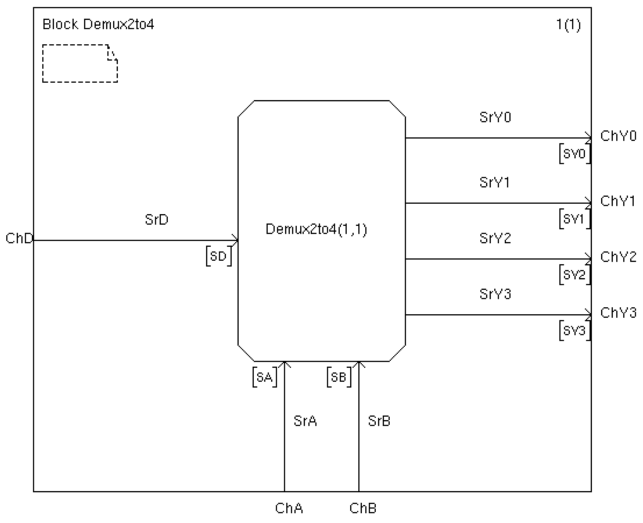


***Figure 18**. Block with a single process to specify a demultiplexer*

As in the previous chapter, outputs are initialised to X (unknown) to avoid inconsistent states during start-up. Once the system is ready, outputs are given values randomly. Different approaches have been followed to initialise devices with valid outputs. As shown in figure 19, an encoder can have any combination of “1s” or “0s” in its output, so the approach on the left has been used. However, a valid output for a demultiplexer consists of certain combinations only, so a solution like the one on the right is better.

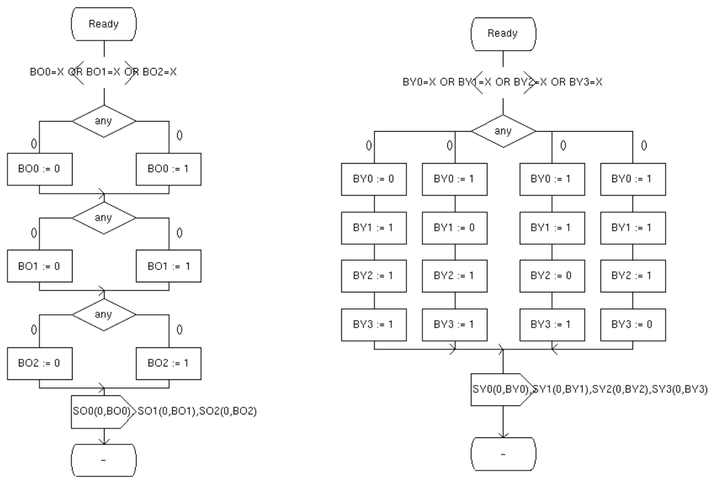
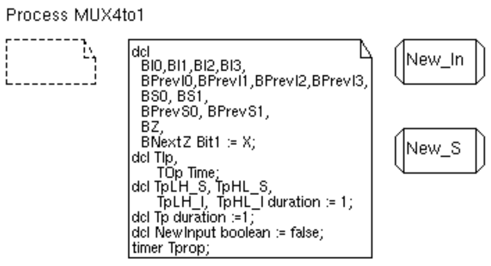


***Figure 19**. Random output initialisation in an encoder (left) and a demultiplexer (right)*

Procedures declared within processes (figure 20) have been extensively used. Some tasks, such as setting the outputs or checking if the inputs have actually changed their values were found repetitive, the only difference being the actual values involved in each instance of the operation. Procedures were very useful, since they can be particularised using different parameters in each call.



```
dcl
  BI0,BI1,BI2,BI3,
  BPrevI0,BPrevI1,BPrevI2,BPrevI3,
  BS0, BS1,
  BPrevS0, BPrevS1,
  BZ,
  BNextZ Bit1 := X;
dcl TIp,
    TOp Time;
dcl TpLH_S, TpHL_S,
    TpLH_I,  TpHL_I duration := 1;
dcl Tp duration :=1;
dcl NewInput boolean := false;
timer Tprop;
```

***Figure 20**. References to procedures* New_In *and* New_S *in the process* MUX4to1

When procedures are referenced within a process, they have access to its variables. Every process has its own variable space, and SDL does not directly allow the use of global variables between processes. However, I could not find any recommendation against accessing process variables in a procedure declared within the process (and the tool certainly did not complain about this). An alternative to this solution could be to pass all process variables that have to be modified as parameters to the procedure. I tried it (I must admit that I felt uncomfortable about accessing variables that were declared outside the procedure), but it made no difference, and in fact it was redundant and not needed at all.

Different solutions have been found to decode inputs and set outputs accordingly. In a BCD-to-Decimal decoder, for example, an internal value is computed to determine the decimal equivalent of the inputs (figure 21). However, in a priority encoder the order in which the inputs are checked is of the utmost importance, so a solution like the one shown in figure 21 (on the right) was chosen. Calls to procedures *Set_Out* contain, as actual parameters, the next output values needed (either 0s or 1s). This does not mean that the outputs are changed instantaneously, as propagation delays have to be taken into account and dealt with properly using timers.

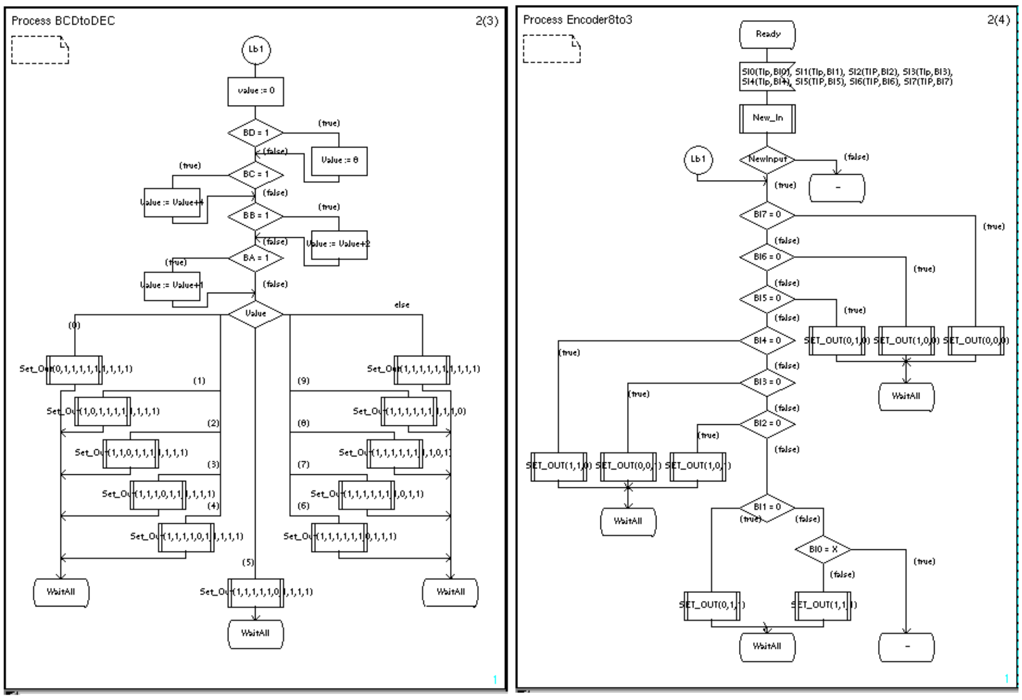


***Figure 21**. Two approaches to input decoding: BCD-to-Dec (left) and Encoder8-to-3 (right)*

In some cases (when there are more than two possible delays in action) the particular delays involved have been used as actual parameters in calls to procedures. In a demultiplexer, for example, the propagation delay between a change in the *Enable* line and the output transition is different to the delay after a change in the *Select* lines. Figure 22 shows a portion of SDL-GR code used to particularise the calls in a demultiplexer. *TpLH_D* and *TpHL_D* are the delays to change the outputs to high and low level after a change in the *Data* line. *TpLH_S* and *TpHL_S* are the ones corresponding to changes in the *Select* lines. These delays are used in the procedure *Set_Out* to run timers.

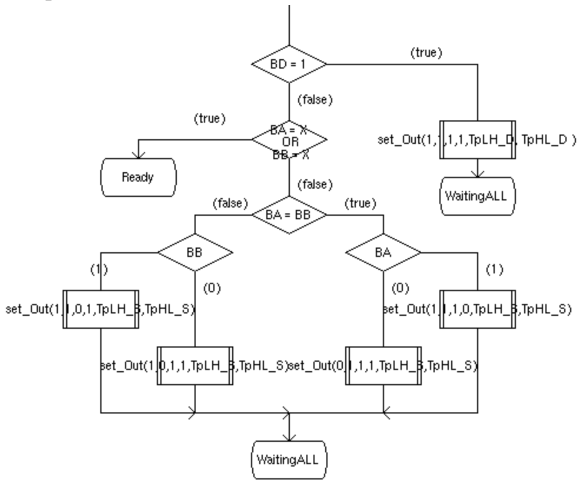

BD = 1
(true)
(false)
(true)
BA = X OR BB = X
set_Out(1,1,1,1,TpLH_D, TpHL_D)
Ready
(false)
WaitingALL
(false)
BA = BB
(true)
BB
BA
(1)
(0)
(0)
(1)
set_Out(1,1,0,1,TpLH_S,TpHL_S)
set_Out(1,1,1,0,TpLH_S,TpHL_S)
set_Out(1,0,1,1,TpLH_S,TpHL_S)
set_Out(0,1,1,1,TpLH_S,TpHL_S)
WaitingALL

***Figure 22**. Use of propagation delays as actual parameters in procedure calls*

The behaviour during the wait states has been described following the approach shown in figure 23. Three different wait states have been used. *WaitingAll* is used to describe the system when the outputs are going to be changed but no one has changed yet. Two different timers are running in this state, *TPropLH* and *TPropHL*. When *TPropLH* expires, the outputs that must go high are changed. If the expiring timer is *TPropHL*, the outputs that must go low are the ones involved in the transition. The procedure *Outputs* deals with the actual output signals being sent. Only one of these timers will expire while in the state *WaitingAll.* Notice that after any timer has expired the system changes its state.

WaitingLH is the state reached after changing the outputs that had to be at low level. Now we are waiting for the exact instant when the other outputs must become high. Only one timer is running in this state, as the other timer expired before entering this state. *WaitingHL* is conceptually similar, but swapping high and low levels as needed.

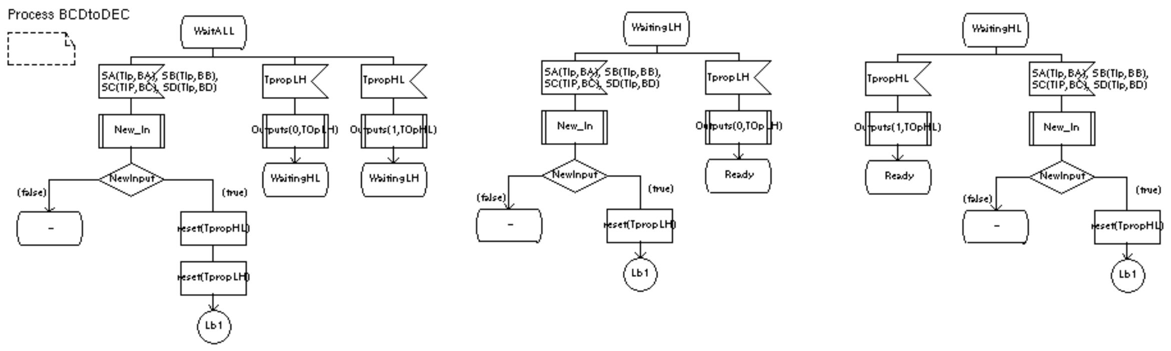


***Figure 23**. Wait states and use of timers to modify outputs at the right time*

If the inputs change their values while the system is waiting, the wait is immediately stopped. The timers that are still running are reset and the new inputs are decoded again as shown in figure 21. After decoding, the next output values and timers are set again.

Three basic types of procedures (with some variants) have been used to specify repetitive tasks. Procedure *Set_Out*, similar to the one shown in figure 24, receives as parameters the next values to be output and, in some devices, the delays that must be used. Using these delays, the times when the outputs will be ready are calculated, and the corresponding timers are set.

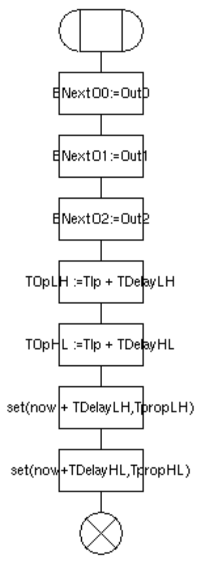


***Figure 24**. Procedure* Set_Out

Several procedures similar to the one shown in figure 25 (for an 8-to-3 encoder) have been used to send output signals. This kind of procedure receives two parameters: the time outputs must be sent at, and the logic level of the outputs that must not be changed in this call. A procedure call like *Outputs(1,TOpHL)*, for example, sends to the environment all output signals that must go to low level at time *TOpHL*. Maybe the other way round could have been more intuitive, in that case a call like *Outputs(1,TOpHL)* would change only the outputs that had to be at high level. The first alternative was used, firstly because it certainly worked, and secondly because it took advantage of a single condition symbol being required, since only those outputs that are not already at the required level are actually changed (both conditions are checked in the same instruction). Notice that a complete change in the state of a device needs two consecutive calls to this procedure. In one call the outputs that change from high to low level are altered. The other call deals with the remaining outputs. The relative values of the high-to-low and low-to-high transition propagation delays determine which ones will be first.

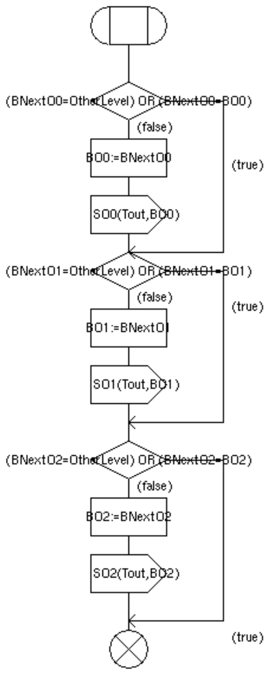


Figure 25. *Procedure* Outputs

5.3 Validation

The SDT validator was again used to test and validate the SDL descriptions of the devices presented in this chapter. Individual components like the ones shown here can be fully validated using the exploration algorithms that were presented in chapter 3. Depending on the complexity of a system, full exploration can take anything from a few seconds to hours or even days. Achieving 100% symbol coverage in the systems presented here was not that time-consuming, but analysing the generated MSCs and performing tests with the navigator certainly was. Some validation options had to be particularised in order to achieve 100% symbol coverage. The maximum depth and abort conditions such as the number of repetitions in the power walk algorithm, for example, had to be increased to achieve 100% coverage in some systems.

MSC traces were used in the analysis of the SDL specifications. They can be viewed as a special trace language, which mainly concentrates on message interchange by communicating entities (such as SDL processes and blocks) and their environment. The main advantage of an MSC is its clear graphical layout, which gives an intuitive understanding of the described system behaviour. Maybe the main disadvantage is that almost all the interpretation work is left to the user. Only some evident errors are reported as such by the tool, but the user has to carefully check that the actual behaviour matches the expectations under every possible combination.

The SDT Validator automatically generates test values for the SDL system to be validated, but the user must also check that the selected values are appropriate to test the system with. When validating a circuit from the ANISEED library, the SDT Validator generates the test values 1, 0, and X for the user-defined sort, Bit1. The X value is unsuitable for input as a test value since this is only used by ANISEED to initialise the inputs and output signals. Fortunately, unsuitable test values can be removed from the list of test values using the ‘clear test value’ option in the validator.

Some initial attempts at fully validating these systems produced some symbol coverages slightly less than 100%. This was further investigated using the coverage viewer, but it sometimes showed that the system actually had 100% symbol and transition coverage. Maybe one reasonable explanation for this could be the use of operators defined in ANISEED's single bit Data Type library (package Bit1). Perhaps the SDT Validator is not able to fully validate this library because the operators have been implemented in C code. However, by trying validation runs with different parameters, 100% coverage was finally reached. It is surprising that the information displayed in the coverage viewer was exactly the same when, say, 98.7% or 100% coverage was achieved-something really strange, I must say. Figures 26 and 27 give some examples of the coverage diagrams that the SDT Validator automatically generates.

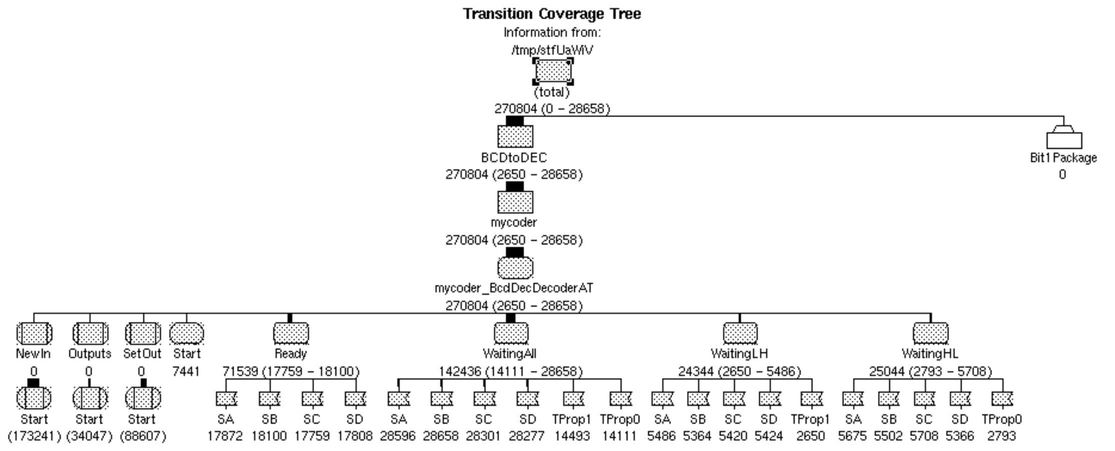


***Figure 26**. Transition coverage tree shown in the coverage viewer (BCD-to-DEC decoder)*

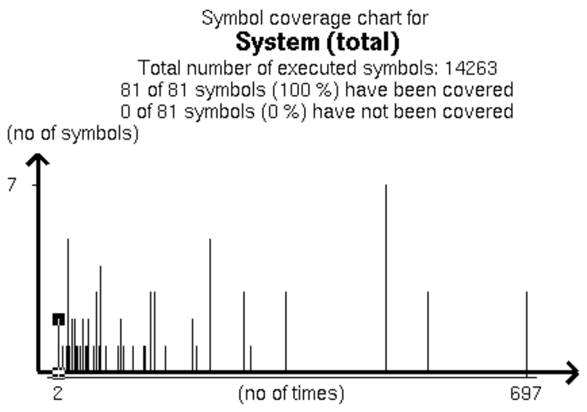


***Figure 27**. Symbol coverage graphs*

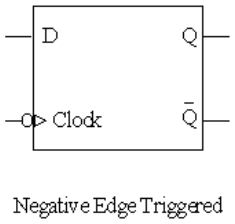
6 Flip-Flops

6.1 Description Issues

Flip-flops are one of the most commonly used devices in sequential circuits. Basically, a flip-flop is a device that can assume one of two stable output states, has a pair of complementary outputs, and one or more inputs that can cause the outputs state to change. There are several kinds of flip-flops, but all have some common characteristics. Some of the most widely used types are the clocked J-K and D (or Delay) flip-flops. These devices react to clock edges (either positive or negative), the output values depending on the inputs. These types, and others such as T (Toggle) or R-S (Reset-Set) flip-flops, are readily available in integrated circuit form.

Different notations can be found in the literature to represent the previous and next states of flip-flops. Previous-state usually means the state of the *Q* output before the active clock edge. Next-state means the state of the *Q* output after the flip-flop has reacted to the clock pulse.

The function table and symbols for two D flip-flops are shown in Figure 28.



D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

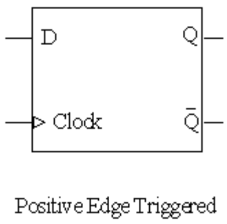
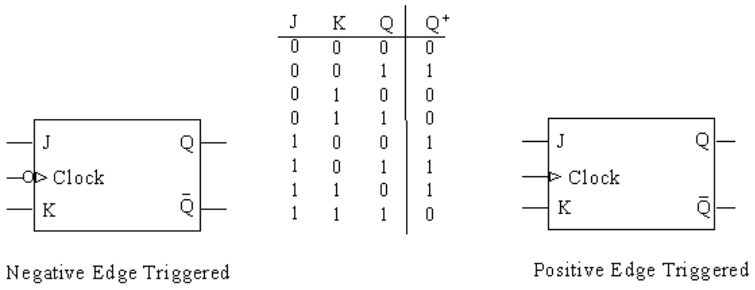


***Figure 28**. Negative and positive edge-triggered D flip-flops*

The state of these flip-flops (Q^+ in figure 28) after the clock pulse is equal to the input D before receiving the clock. For example, if D = 1 before the clock pulse, Q will be 1 after the clock edge regardless of the previous value of Q. The arrowhead on the D flip-flop symbol marks the clock input, and the small inversion circle indicates that the state changes occur on a high to low transition (negative-edge triggering). When there is no inversion circle (as in the right side of figure 28) the state changes occur on a low to high transition (positive-edge triggering).

A clocked J-K flip-flop (figure 29) has three inputs: J, K and the clock. This component changes state a short time after the rising or falling edge of the clock pulse (depending on the kind of device). If J = 1 during the clock edge, *Q* will be set to 1. If K = 1 during the clock pulse, *Q* will be set to zero. However, if J = K = 1, *Q* will toggle state after the clock active edge. If J = K = 0 the outputs will remain the same. The change in state is initiated by the clock pulse and never by a change in J or K.



J	K	Q	Q^+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

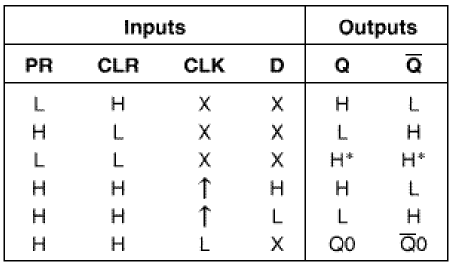
***Figure 29**. J-K flip-flops (negative and positive-edge triggered)*

A J-K flip-flop is more versatile than a D flip-flop. Only two operations are possible with the D flip-flop: setting the D flip-flop output to 1, and resetting its output to 0. Four operations are possible with the J-K flip-flop. Besides the operations of setting or resetting its output at each clock transition, the J-K flip-flop may also toggle or remain in the same state.

Flip-flops have some important timing constraints and characteristics that must be considered when analysing sequential circuits. In a D flip-flop there are two main considerations. Firstly, when the clock makes the active transition the outputs do not change instantaneously: there is a certain propagation delay between these two events. The second consideration is that the data on the D input should be steady before the clock pulse. If the data is changing too closely to the instant of the clock pulse the stored value is unpredictable. For this reason, the setup time is the minimum time interval the input must be stable before a clock pulse. Similarly, the hold time is the minimum time interval the input must be held steady after the clock edge. If these timing constraints are not respected the flip-flop output is unpredictable.

Users of J-K flip-flops also have to take timing characteristics into consideration. Both the J and K inputs have associated time intervals, t_{Setup} and t_{Hold}, where t_{Setup} is the minimum time interval the J and K inputs must be stable before the clock pulse and t_{Hold} is the minimum time the inputs must be held steady after the clock pulse. Failure to adhere to these timing constraints again results in an unpredictable output.

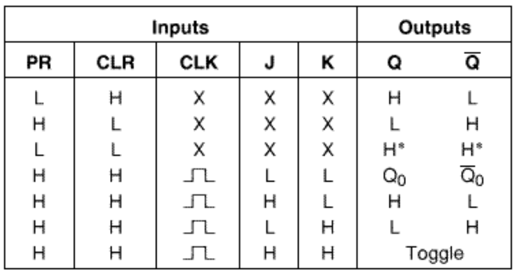
Integrated circuit flip-flops often have additional inputs (*Clear* and *Preset*) that can be used to set the flip-flop to an initial state independently of the clock. An appropriate logical level applied to the *Clear* input will reset the flip-flop to *Q* = 0 and *Qbar* = 1. Similarly, an active signal on the *Preset* input will set the flip-flop to *Q* = 1 *Qbar* = 0. These inputs override the clock and any other input. That is, a signal applied to the *Clear* input will reset a J-K flip-flop regardless of the values of J, K and the clock. As an example, figure 30 shows the function table for a D flip-flop with low level active *Preset* and *Clear* (54HC74A). In this figure H represents a high logic level, L a low level and X is either high or low level. *Q0* is the level of the *Q* output before the indicated input conditions were established. The states marked with an asterisk represent non-stable configurations; that is, they will not persist when *Preset* and *Clear* return to their inactive level. The arrows represent positive clock edges.



Inputs				Outputs	
PR	CLR	CLK	D	Q	$\bar{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q0	$\overline{Q0}$

***Figure 30**. Positive-edge triggered D flip-flop with* Preset *and* Clear

Master-slave versions of the flip-flops discussed above are also commercially available. These devices need a complete clock pulse (with a rising edge and a falling edge) to change their outputs. For example, a master-slave J-K flip-flop (figure 31) processes the J and K data after a complete clock pulse. While the clock is low the slave is isolated from the master. On the positive transition of the clock, the data from the J and K inputs is transferred to the master. On the next negative transition of the clock, the data from the master is transferred to the slave. The logic state of J and K inputs must not be allowed to change while the clock is high. As in previous flip-flops, an active logic level on the *Preset* or *Clear* inputs will set the outputs regardless of the other inputs.



Inputs					Outputs	
PR	CLR	CLK	J	K	Q	$\overline{Q}$
L	H	X	X	X	H	L
H	L	X	X	X	L	H
L	L	X	X	X	H*	H*
H	H	⎍	L	L	Q_0	$\overline{Q}_0$
H	H	⎍	H	L	H	L
H	H	⎍	L	H	L	H
H	H	⎍	H	H	Toggle	

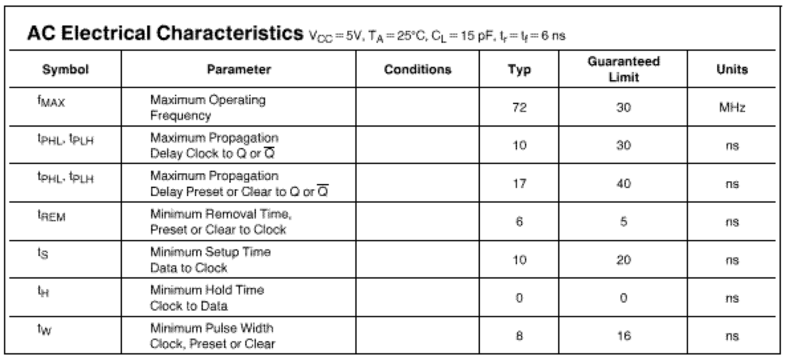
Figure 31. *Function table for a master-slave J-K flip-flop with* Preset *and* Clear

In figure 31, the positive pulse symbol in the clock column indicates that it is a master-slave flip-flop, so J and K must be held constant while the clock is high. As commented above, data is transferred to the outputs on the falling edge of the clock pulse. Toggle means that each output changes to the complement of its previous level on each complete positive clock pulse.

6.2 Library Components

Twenty different models of flip-flops have been specified in SDL, validated and included in the new ANISEED library. Appendix B includes details of the devices available and their main characteristics. Due to space limits, only some general issues and a small example will be described here.

Timing constraints in flip-flops proved rather complicated to specify, especially in models with *Preset* and *Clear* inputs. As shown in figure 32, a typical D flip-flop with *Preset* and *Clear* includes parameters such as the maximum operating frequency, setup and hold times, 4 different propagation delays, removal (or recovery) times and minimum pulse widths.



AC Electrical Characteristics $V_{CC} = 5V$, $T_A = 25°C$, $C_L = 15$ pF, $t_r = t_f = 6$ ns

Symbol	Parameter	Conditions	Typ	Guaranteed Limit	Units
f_{MAX}	Maximum Operating Frequency		72	30	MHz
t_{PHL}, t_{PLH}	Maximum Propagation Delay Clock to Q or $\overline{Q}$		10	30	ns
t_{PHL}, t_{PLH}	Maximum Propagation Delay Preset or Clear to Q or $\overline{Q}$		17	40	ns
t_{REM}	Minimum Removal Time, Preset or Clear to Clock		6	5	ns
t_S	Minimum Setup Time Data to Clock		10	20	ns
t_H	Minimum Hold Time Clock to Data		0	0	ns
t_W	Minimum Pulse Width Clock, Preset or Clear		8	16	ns

Figure 32. *Timing parameters for a D flip-flop with* Preset *and* Clear *inputs*

The problem of formally specifying these devices in SDL was initially tackled following a one-block/one-process approach. This solution was rather straightforward for D or T flip-flops without *Preset* or *Clear* inputs, but it proved inadequate for more complicated devices. Using just a single process led to very complex descriptions with a large number of possible states and very intricate timing behaviour. The resulting specifications were difficult to understand and, what was even worse, adding new elements to complete a specification required further and error-prone changes in all previous parts. It seemed clear that a multi-process approach was needed, so solutions similar to the ones shown in figure 33 were used.

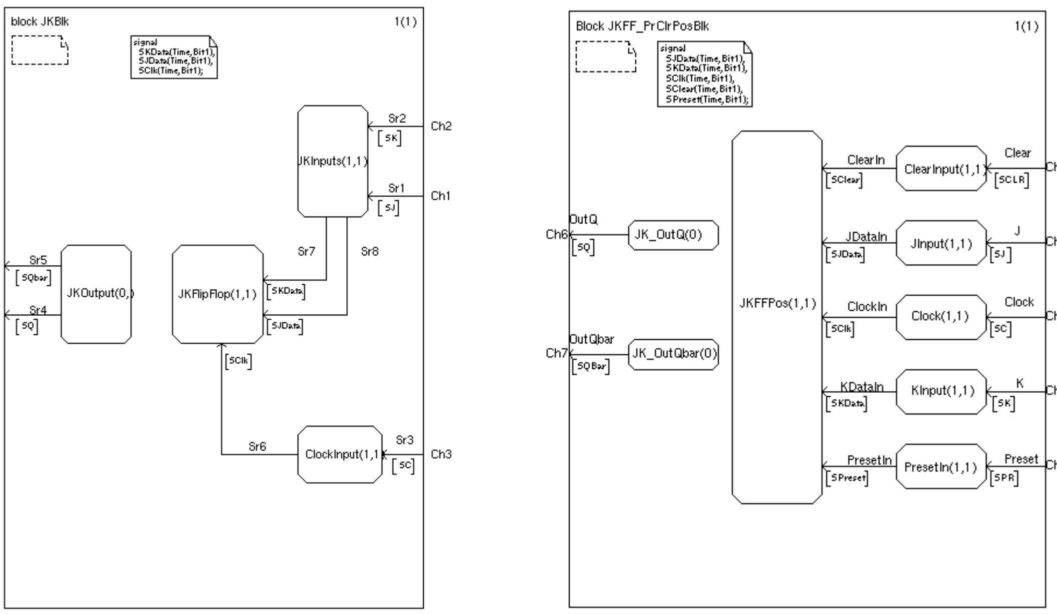


Figure 33. *Multi-process SDL descriptions for flip-flops with different complexity levels*

The diagram on the left side of figure 33 shows one possible solution for a J-K flip-flop without *Preset* or *Clear*. Both inputs (J-K) are included in a single process named *JKInputs*. The clock signal goes to another process that deals with the maximum frequency rate for the device. Both outputs (*Q* and *Qbar*) are also included in a single process *JKOutput*. Multiple instances of this process can be created by the process *JKFlipFlop* as required.

The diagram shown on the right side of figure 33 corresponds to a J-K flip-flop with *Preset* and *Clear*. This is one of the most sub-divided specifications that have been developed. It contains one process for every input signal. Even the two outputs are dealt with separately, the reason for that being that now they must be controlled independently. In this device *Qbar* is not always the negation of *Q* since there is a non-stable state (when both *Preset* and *Clear* inputs are active) in which both outputs are at a high level (see figure 31). A different solution consisting of a single output process with independent parameters for the two output values was also tested in other flip-flop specifications.

With these multi-process approaches some additional internal signals are required. In the diagrams shown in figure 33 only the internal signals are declared. The obvious reason for this is that the external ones were declared with the external block (not included in the figure). The central process *JKFFPos* receives these internal signals from the input processes without having to be bothered about external signal timing constraints. Now, the particular timing aspects for each input signal are dealt with in its corresponding process. Setup times, minimum pulse widths and maximum operation rates, for example, are considered in these processes. Notice that, as shown in the figure, input processes have one initial instance and can have a maximum of one instance, while the output processes have no initial instances and can have an infinite number of instances.

Timers are used to model the time difference between input signals arriving and the output being generated. Several timing aspects must be taken into consideration. First of all, upon receiving an input the corresponding setup timer is set. For data inputs this setup timer represents the time prior to receiving a clock pulse that the signal must be present. In *Preset* or *Clear* inputs the setup time can be used to model the minimum pulse width required for these signals (the flip-flop does not react to *Preset* or *Clear* active signals shorter than a certain duration). New inputs during the setup time make the process re-start the corresponding timer and re-enter the setup state. Only when an input signal has completed its setup time is it made available to the central process *JKFFPos.* As an example, figure 34 shows an (incomplete) SDL specification for the process *PresetIn.* Only when the timer *TPulseMin* expires is the internal signal *SPreset* sent (at the calculated time *TOp*) to the central process. *Preset* inputs shorter than the minimum time given as parameter will not cause the flip-flop to react.

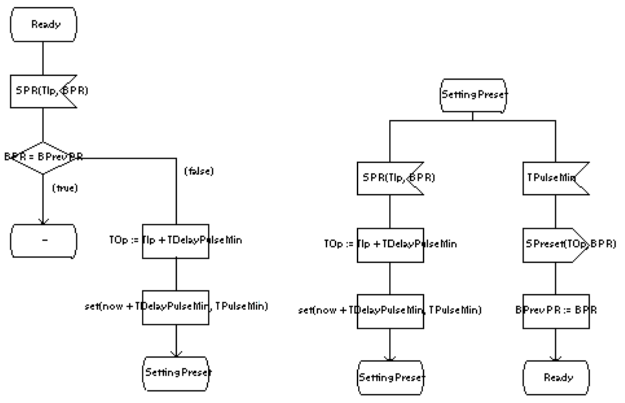


Figure 34*. Minimum pulse width specification in the* Preset *input*

Similarly, the process *Clock* (figure 35) uses a timer to avoid the flip-flop working above its maximum frequency rate.

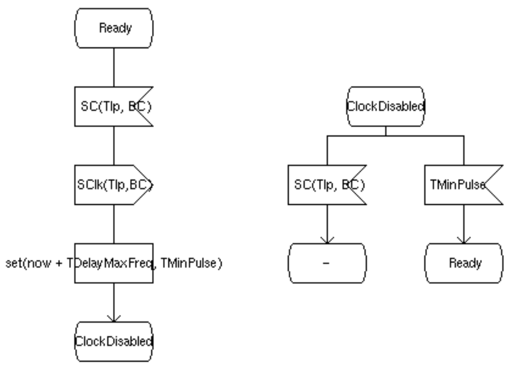


Figure 35*. Avoiding the flip-flop being overdriven if the clock is too fast*

As shown in the figure, after a clock edge has been received there is a period of time where no new clock pulses can possibly be attended to. A flip-flop working at clock speeds faster than the nominal rate will certainly behave in a strange way. Even physical damage to the device might result, but no explicit information about this issue is given in datasheets. The SDL specification presented in figure 35 deals with this problem by ignoring clock pulses faster than the nominal rate. With this approach, the flip-flop will simply ignore any premature clock pulse.

Similar to the devices presented in previous chapters, the central process *JKFFPos* randomly initialises the flip-flop outputs during startup. As shown in figure 36, when *BQ* (the value of output *Q*) is unknown (X), instances of the output processes are created with a random value and time 0 as parameters. The same value (*BNextQ*) is passed to the two output processes. However, as will be described later, the process *JK_OutQbar* internally negates this parameter.

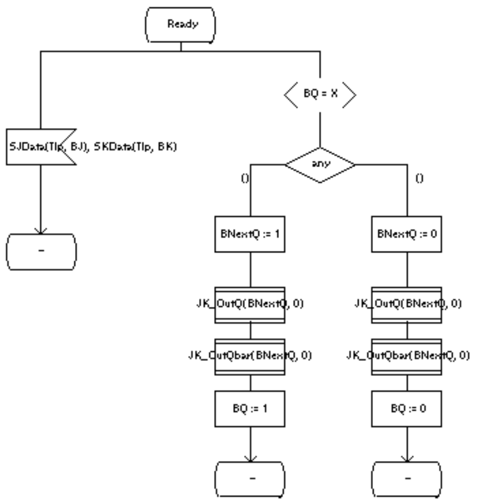


***Figure 36**. Random output initialisation during startup*

Figure 36 also shows that new internal data signals do not cause the flip-flop to change its state. This flip-flop only reacts to positive-edge clock signals, so the new inputs (after having finished their setup time) are just received without further processing until the appropriate clock signal arrives.

Figure 37 shows the behaviour of the flip-flop when *Preset* or *Clear* signals are received while the flip-flop is in the state *Ready*. First of all, the value of the signal is checked. *Preset* or *Clear* signals in this particular flip-flop are low-level active, so only when their values (*BCLR* or *BPR*) are 0 are further actions needed. If the outputs have to be modified, new instances of the output processes are created with the proper values and times as parameters. Depending on the input signal received, the system changes either to the state *PreSetting* or *Clearing*.

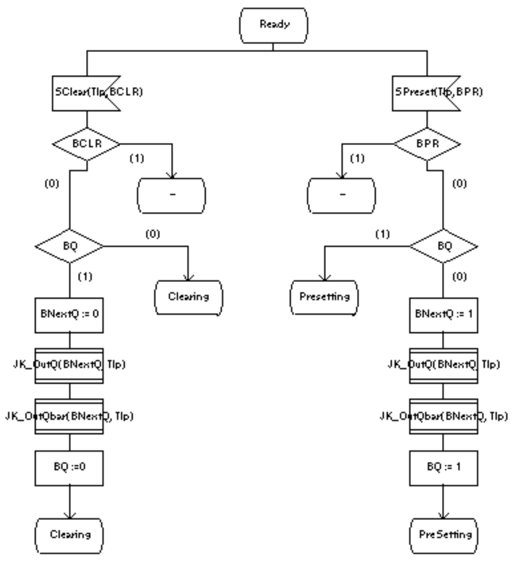


***Figure 37**.* Preset *or* Clear *signals while in the* Ready *state*

The decoding of inputs after receiving a positive clock edge is shown in Figure 38. Basically, four different paths can be taken. If the inputs J and K are both 0, no change in outputs is required so the flip-flop is ready again without further actions. If J and K are both 1, the flip-flop must toggle its outputs so a timer (*Thold)* is set to control the hold time before entering the state *Holding_TQ*. Similarly if J = 1 and K

= 0, the flip-flop enters the state *Holding_SQ* (after the hold time the output *Q* will be set and *Qbar* cleared). Finally when J = 0 and K = 1, the flip-flop enters the state *Holding_RQ* to indicate that the output *Q* is going to be cleared and *Qbar* set to 1. Notice that the inputs that are decoded are the internal data signals, not the external ones that come from the environment. These internal signals are considered steady since they were generated after the external ones finished their setup times.

Some time was spent analysing the correctness of this approach, since bizarre situations may occur. For example, imagine that the external inputs J and K are received, their setup times completed, and the corresponding internal signals are sent to the process *JKFFpos*. This process is not concerned about setup times, so as soon as it receives a positive clock pulse it decodes the values of the internal data signals to set the outputs accordingly. Everything goes fine so far, but what would happen if during the hold time a new external input were received? Setup times are usually longer than hold times, so the central process would not know that the external inputs have changed while it was doing the holding. The outputs would be set as appropriate and everything considered to be finished. Only when the external data signals finished their setups would the central process know the new values, but it would not react to them until a new clock pulse was received. This flip-flop specification then seems to have some sort of inertia. It does not respond to input changes until setup times have finished. In an extreme bizarre situation where the inputs changed continuously without finishing their setups, the central process (and hence the whole flip-flop) would remain ignorant of the external events. However, this situation (and some others that caused concern while developing the flip-flop specifications) are examples of non-deterministic behaviour. If setup times are not respected the flip-flop will behave in a way that is not specified by the manufacturer. Datasheets, and hence the SDL specifications, only guarantee that when the timing constraints are respected will the system behave as predicted. If for some reason setup times, hold times or clock rates are violated the behaviour of the device will be unpredictable. Bearing this in mind, it does not really matter what output values the flip-flop has when timing is not respected. Any logic level would be defensible, since the device is not working under its normal conditions.

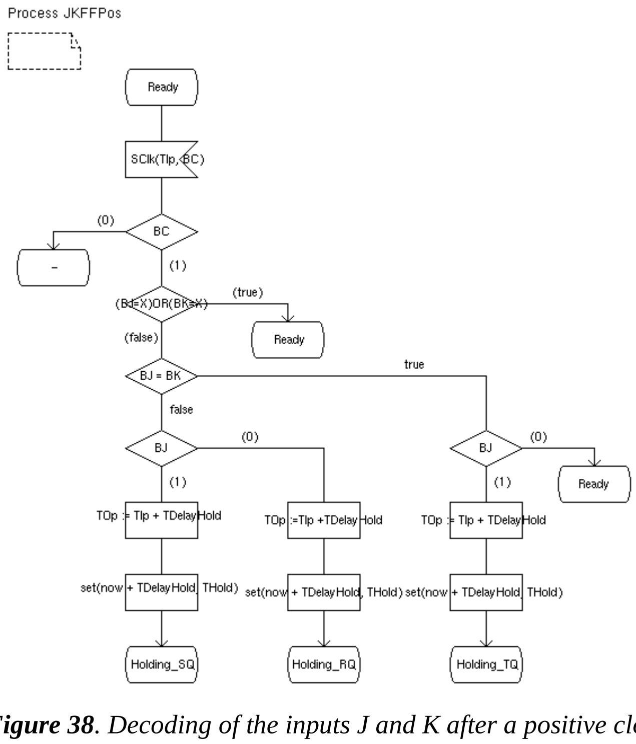


***Figure 38**. Decoding of the inputs J and K after a positive clock edge*

Preset and *Clear* signals can also be received while the flip-flop is in one of the three possible holding states. As shown in figure 39, when this happens the active hold timer is reset, the output processes are instantiated, and the system is moved from the original holding state into *PreSetting* or *Clearing.*

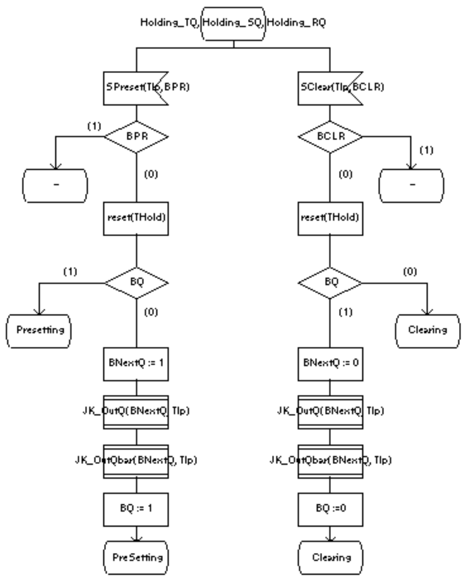


***Figure 39**. Preset or Clear signals during hold time*

New clock pulses are simply ignored during hold time (figure 40). This situation is not specified in datasheets and it is very unlikely to occur in practice, since hold times for modern flip-flops are usually as short as one nanosecond. Two consecutive clock pulses in less than one nanosecond is clearly beyond the normal frequency rate for common flip-flops. This situation would also be intercepted in the process *Clock* if the parameter "maximum clock rate" were set to a sensible value. Figure 40 also shows that new data signals *SJData* and *SKData* cause the holding time to be interrupted and the system moved to the state *Ready*.

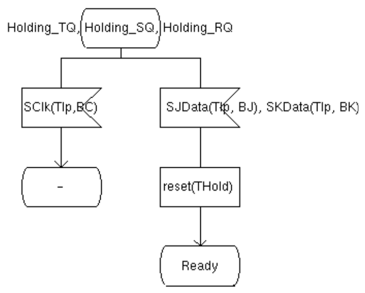

Holding_TQ, Holding_SQ, Holding_RQ
SClk(Tlp,BC)
SJData(Tlp, BJ), SKData(Tlp, BK)
-
reset(THold)
Ready

***Figure 40**. New clock pulses or data signals during hold time*

Figure 41 shows that new output processes are created when the timer *THold* expires. This figure corresponds to the state *Holding_RQ*, but similar specifications have been used for the other two hold states (*Holding_SQ* and *Holding_TQ*).

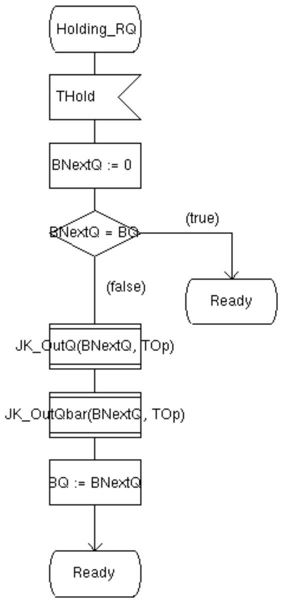


Figure 41. Output *process instances after a hold timer has expired*

The behaviour of the flip-flop in the states *Clearing*, *Presetting* or *ClearAndSet* (this state is reached when the *Preset* and *Clear* inputs are both active) is very similar. As an example, figure 42 shows the specification for the state *Clearing*. In this state the flip-flop only reacts to *Preset* or *Clear* inputs, either to return the device to its *Ready* state or to move it to *ClearAndSet* when both are active at the same time. New data inputs or clock pulses are ignored in this state. The *ClearAndSet* and *Presetting* states, not shown here, are similar.

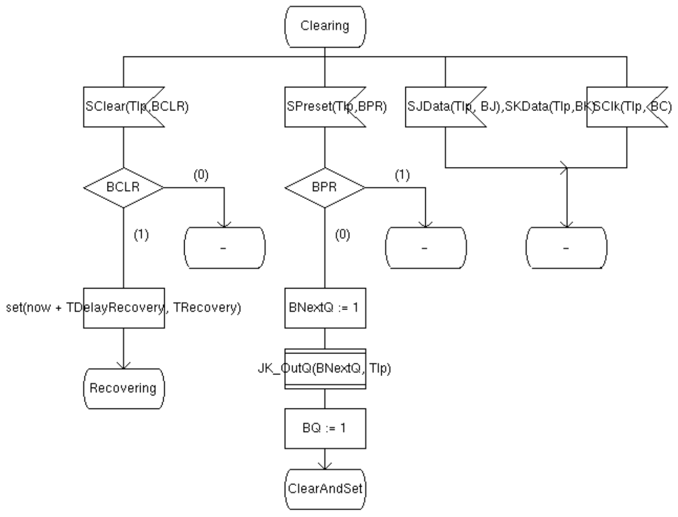


Figure 42. Clearing *state behaviour*

As described in figure 32, a flip-flop is not immediately ready after a *Preset* or *Clear*. When these signals are inactive again, there is a removal or recovery time during which no clock pulses can possibly be attended. This has been modelled in SDL using another timer (*TRecovery*) and the diagram shown in figure 43. During recovery new clock pulses and data signals are ignored. However, *Preset* and *Clear* are independent of the clock and, for that reason, can make the device to enter into *Clearing* or *Presetting* again. If the timer *TRecovery* expires, the flip-flop enters the state *Ready* so new clock edges will be detected again.

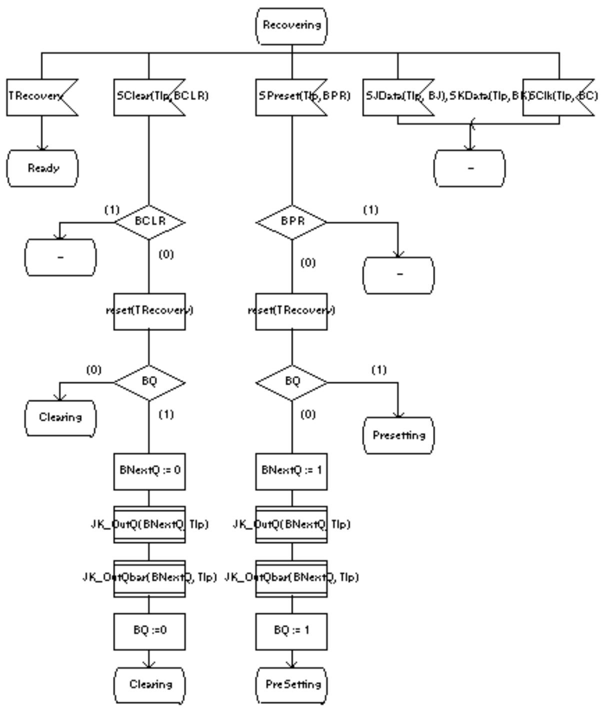


***Figure 43**. Recovery time after finishing a* Preset *or* Clear

Finally, one of the two output processes (*JK_OutQbar*) is shown in figure 44. This process calculates the actual output time as the input process time plus the propagation delay duration (*TDelayProp*). It also sets a timer *TProp* that models the time it takes for the output to be generated. Once the *TProp* timer expires, the output signal *SQBar* is generated. Notice that this particular example uses a bit operator *NotB* to invert the value *BQ* passed as a parameter. After sending the output signal this output process instance dies, but a new one will be created later if a new output is required.

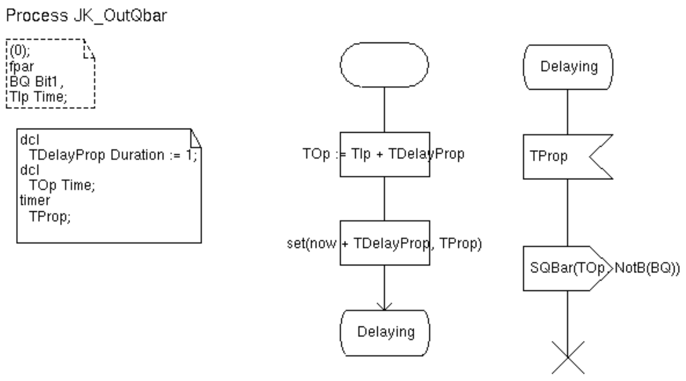
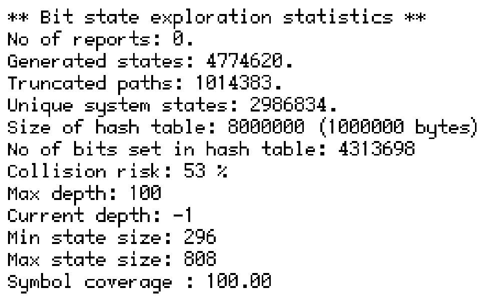


***Figure 44**. Output process showing the formal parameters*

6.3 Validation

Except for the simplest T flip-flop specifications, exhaustive exploration for the flip-flops presented here has not been possible. The state space explodes exponentially and the SDT Validator terminates early with an 'out of memory' error message. A valid alternative has been to use the random-walk or power-walk algorithms with the SDT Validator advanced options set.

Bit state explorations for these devices achieved 100% symbol coverage (figure 45). The time needed strongly depends on the complexity of the description. Sometimes the default parameters of the SDT Validator had to be changed. Values such as the termination conditions in the power walk algorithm (number of repetitions, maximum depth, etc.) had to be increased to achieve 100% symbol coverage.



```
** Bit state exploration statistics **
No of reports: 0.
Generated states: 4774620.
Truncated paths: 1014383.
Unique system states: 2986834.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 4313698
Collision risk: 53 %
Max depth: 100
Current depth: -1
Min state size: 296
Max state size: 808
Symbol coverage : 100.00
```

Figure 45. *Bit state exploration for a D flip-flop*

Besides automatic exploration, MSCs have been extensively used to analyse behaviour. They have proved useful to find intricate errors and strange combinations that caused problems. As an illustrative example of the kind of things MSCs are useful for, figure 46 (left) shows a sequence chart taken from an originally wrong specification. Here, a *Preset* signal has finished its setup time while the flip-flop is recovering from a previous *Preset*. This specification ignored *Preset* signals during recovery, but that was certainly wrong. Since *Preset* and *Clear* are independent level signals, the flip-flop should immediately enter the state *Presetting* again. This situation is very unlikely to occur in physical flip-flops, since recovery times (typically 6 ns) are usually shorter than *Preset* or *Clear* setup times (typically 9 ns). The diagram on the right shows what would happen in a flip-flop with sensible parameters; now the behaviour is correct. However, just in case flip-flops with strange parameters exist, the original specification was modified to get round this problem.

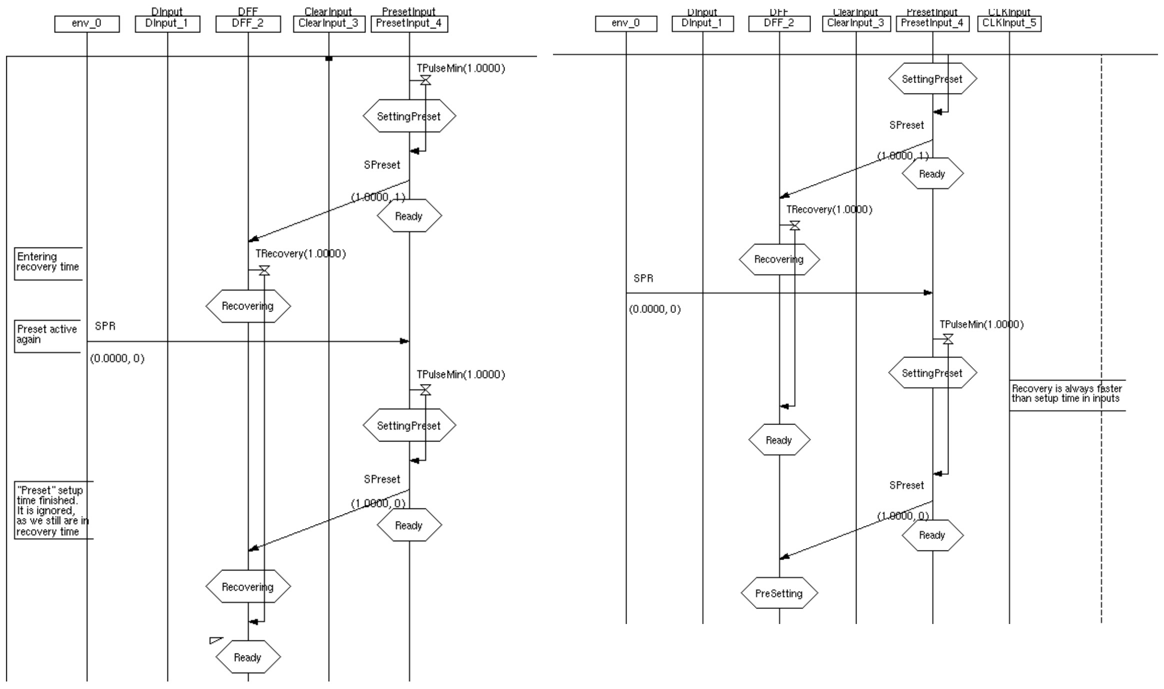


Figure 46. *Use of Message Sequence Charts to analyse system behaviour*

7 Putting It All Together

7.1 Constructing the New ANISEED library

The SDL specifications presented in the previous chapters were not added to the library in their original format. Although it would have been possible to specify all the library components individually, this would have been very inefficient. For example, a two-input tri-state *Nand* gate has largely the same specification as one with three, four or eight inputs. The gates for *And*, *Or*, *Nor*, *Xor* and *Xnor* differ from *Nand* only in their logic function. Since each kind of logic gate has uni-bit/multi-bit and untimed/timed versions, a large number of variants would have to be specified explicitly. As a more pragmatic solution, these were all generated automatically from an SDL template that is parameterised by the logic function, the number of inputs, whether timed and whether multi-bit.

The original SDL-GR specifications were converted into textual SDL-PR format and comments added to the code. This proved a time-consuming and error-prone task, as thousands of lines of SDL code were generated. After adding comments, the resulting PR files were re-imported into the tool, trying to re-construct the original SDL-GR systems. That was the best way of checking that the final PR files (after all the editing work) were still correct. Common problems were minor syntactic errors due to the editing.

Using the SDL-PR files, Ken Turner constructed library modules in the macro language *m4* format. These files were processed by the macro processor *m4* [25] to generate the new PR library packages. This approach makes the *m4* library much smaller and more maintainable, since a single template needs to be changed if modifications are required. In every case a library PR package is automatically generated from a library *m4* module, including explanatory comments.

The conversion from the original PR files to library form required many global edits and modifications of a syntactic nature. All original blocks became block types. Block types were given synonym context parameters for timing parameters and signal context parameters for external signals. Block types were also given gate parameters corresponding to channels. Processes needed *SignalSet* parameters for inputs. All these modifications and global edits proved a real risk of accidental alterations.

The next step in constructing a reliable library was to instantiate and test the brand new block types. We constructed SDL systems with single blocks that were instances of the block types contained in the library. Due to limitations of the SDT tool, a filter written by Ken Turner was used to handle context parameters. An environment variable (*SDLPATH*) causes an automatic search for the corresponding definitions. The only packages that need explicit " #INCLUDE " or "Use" clauses are those that have to be imported literally (basically the "Bit1" and "BitM" packages for single and multiple bits). Appropriate parameters are given and gates connected via channels to the environment. The signalroutes within the block (i.e. among its processes) are automatically inferred by SDT.

The resulting systems were tested again as before they were put in the library. Unfortunately, due to time limitations, a thorough and complete validation could not be performed again for all block types. However, they were instantiated to check that SDT could use them without errors. Whenever problems were found, they were corrected in the *m4* source files, and the library reconstructed again. The most common errors were wrong number of parameters in block types, missing gate declarations, missing *SignalSets* and some spelling problems with signal names.

Further problems with SDT arose while trying to use the library to do some case studies. Context parameters are not supported by SDT so they are dealt with by a script (*sdlctx*) developed by Ken Turner and used as a filter in the SDT analyser. The interactions between the script and the tool are currently being investigated by Ken Turner, but error messages about several units leading to the same name were given when trying to start a validator in systems that included block type instances (Telelogic has subsequently confirmed that this is due to a limitation of SDT). Fortunately, a temporary solution to get round the problem was found. Since no errors were reported by the tool when running the analyser (only the validator was in conflict), a valid PR file was generated by SDT for each of our systems. Converting these PR files into GR format and importing them into the tool was a messy but usable solution to validate systems with instantiated block types. Following this approach, the ANISEED library could finally be used to implement and analyse electronic designs. The new available elements in ANISEED library are summarised in appendix B.

Achieving 100% symbol coverage when validating instances of block types proved harder than with the original descriptions. The selection of parameters for block types is critical. Depending on the values given to parameters such as propagation delays, some parts of the system may not be covered. This is something reasonable, since the specification contains descriptions for all possible situations. For example, if a particular component has a propagation delay for low-to-high transitions (*TpLH*) shorter than *TpHL*, (high-to-low) the portion of the general specification that deals with *TpHL* finishing first will never be executed. Several validation runs with different parameters were tried and, sometimes, conflicting parameters were given the very same values, so that the validator could follow all possible paths. Even with this approach some devices such as flip-flops with preset and clear signals needed long validation times.

7.2 Using The New Library

Once the library is ready to use, modelling electronic circuits in SDL is a rather straightforward task. A circuit can be specified as a SDL system with some inputs and outputs to interact with the environment. The system will be composed of SDL blocks that are instances of the block types included in the library.

As mentioned before, all the elements included in the library are generic block types that can be instantiated with actual parameters. The values given to these parameters allow designers to adapt the behaviour of the circuits under test to their particular needs. Parameters such as propagation delays, setup intervals etc. are readily available in datasheets, making it possible to represent different logic families and commercial devices.

The SDL-PR files contained in the ANISEED library include details about the parameters needed to instantiate every particular block type. For example, as shown in the following fragment taken from *aniseed_mux.pr*, to instantiate a four-to-one multiplexer four timing parameters are needed: *TDelayS1*, *TDelayS0*, *TdelayD1* and *TdelayD0*.

```
Block Type FourOneMultiplexerAT
       <
        Synonym TDelayS1 Duration;                 /* Select-high prop. delay */
        Synonym TDelayS0 Duration;                 /* Select-low prop. delay */
        Synonym TDelayD1 Duration;                 /* Data-high prop. delay */
        Synonym TDelayD0 Duration;                 /* Data-low prop. delay */

        Signal                                     /* Input-output signals */
         SIp0 (Time, Bit1), SIp1 (Time, Bit1),     /* Inputs 0-1 */
         SIp2 (Time, Bit1), SIp3 (Time, Bit1),     /* Inputs 2-3 */
         SSel0 (Time, Bit1), SSel1 (Time, Bit1),   /* Select 0 and Select 1 */
         SOp (Time, Bit1)                          /* Output */
       >;
```

TDelayS1 and *TDelayS0* represent the propagation delays after a variation in the *Select* inputs. Two values are needed since these delays are usually different depending on the output previous and next logic levels. *TDelayS1* is the delay for low-to-high output transitions, and *TDelayS0* the delay for high-to-low ones. Similarly, *TDelayD1* and *TDelayD0* represent the low-to-high and high-to-low propagation delays after a variation in the multiplexer *Data* inputs.

If a particular four-to-one multiplexer has to be used in a circuit, now it can be modelled as an instance of the ANISEED library block type *FourOneMultiplexerAT.* The name for the new block, the block type to be instantiated, the parameters and the signals involved must be given. As an example, the following code creates a instance with timing values (in nanoseconds) taken from a datasheet:

MyMux:FourOneMultiplexerAT<18, 27, 18, 24, SIp0, SIp1, SIp2, SIp3, SSel0, SSel1, SOp>

MyMux is the name given to the new block. Any valid SDL name can be selected for this purpose. The semicolon indicates that *MyMux* is an instance of a block type, in fact it is an instance of the block type *FourOneMultiplexerAT* included in the library. The four numerical values are the actual parameters in nanoseconds for the propagation delays (*TDelayS1*, *TDelayS0* etc). *SIp0* to *SIp3* are the *Data* input signals, *SSel0* and *SSel1* the *Select* input signals and *SOp* the multiplexer output signal. These signals must also be declared in the SDL diagram.

Telelogic SDT does not support context parameters, so they are managed by a script (*sdlctx*) written by Ken Turner and used as a filter in *SDT*. Every device in the library needs some particular parameters to be instantiated. For example, as shown in the following PR code taken from *aniseed_flipflop.pr*, a D flip-flop without *Preset* or *Clear* only has three parameters: the *Setup* period, the *Holding* time and the propagation delay. If the number of parameters passed in the call is not consistent with the block type declaration, *sdlctx* reports the problem.

```
Block Type DFlipFlopAT
       <
        Synonym TDelaySetup Duration;              /* Setup delay */
        Synonym TDelayHold Duration;               /* Hold delay */
        Synonym TDelayProp Duration;               /* Prop. delay */
        Signal                                     /* I/O signals */
          SC (Time, Bit1), SD (Time, Bit1),        /* Clock and Data */
          SQ (Time, Bit1), SQBar (Time, Bit1)      /* Outputs */
       >;
        Gate D In;                                 /* Input data */
        Gate C In;                                 /* Input clock */
        Gate Q Out;                                /* Output */
        Gate QBar Out;                             /* Output negated */
```

As shown in the PR code above, besides the timing parameters four signals are also needed to instantiate a D flip-flop: two inputs (*Clock* and *Data*) and the outputs. The corresponding gate declarations for these signals are also shown in the code. Since every block type instance includes the input and output signals, the interconnecting wires for the circuits in SDL diagrams are actually optional. However, they can be modelled as SDL channels for clarity. *Sdlctx* also infers the gate names, so they do not have to be explicitly typed in the SDL diagram. To make things clearer, the following section presents an example of a complete SDL system with two instances of D flip-flops and other components. Some points presented here are discussed in more detail.

7.3 Case Study – The Single Pulser

In order to demonstrate that the ANISEED methodology can be used to implement and analyse practical circuits, a case study was carried out. This case study is a circuit design taken from the standard benchmark circuits for verification and validation [28]. These benchmark circuits are widely used to assess the performance of hardware validation tools.

A single pulser is a clocked-sequential device with one-bit input *I* and one-bit output *O*. It has a debounced push-button, on (true) in the down position, off (false) in the up position. An electronic circuit senses the depression of the button and asserts 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. Assuming that the circuit is synchronous, the specification may be stated as saying that for each input pulse, the Single Pulser issues exactly one pulse of unit duration regardless of the duration of *I*. The specification may be also characterised by the following three properties [29]:

1. Whenever there is a rising edge at *I*, *O* becomes true some time later.

2. Whenever *O* is true it becomes false in the next time instance and it remains false at least until the next rising edge on *I*.

3. Whenever there is a rising edge, and assuming that the output pulse does not happen immediately, there are no more rising edges until that pulse happens (There can not be two rising edges on *I* without a pulse on *O* between them).

The implementation shown in figure 47 is taken from [30]. The incoming, not yet debounced asynchronous signal ***Pulse_In*** is fed to a D flip-flop and thus becomes the synchronised signal ***Pulse_sync***, which is then delayed for one clock cycle by using another D flip-flop. Its output is negated, and the And-connection of the synchronous pulse with its own delay generates the resulting, one clock-cycle lasting signal ***Pulse_Out***. Figure 48 shows some ideal (no delays) waveforms that illustrate the behaviour of the circuit.

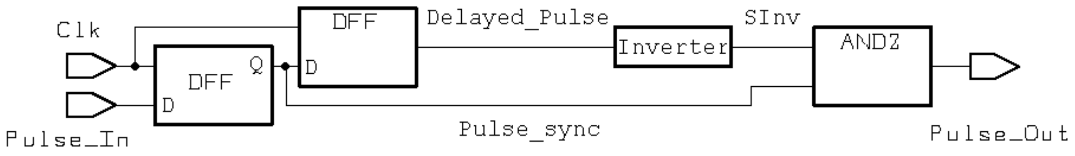


***Figure 47**. One possible implementation of the single pulser*

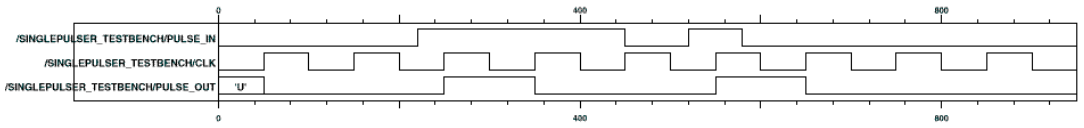
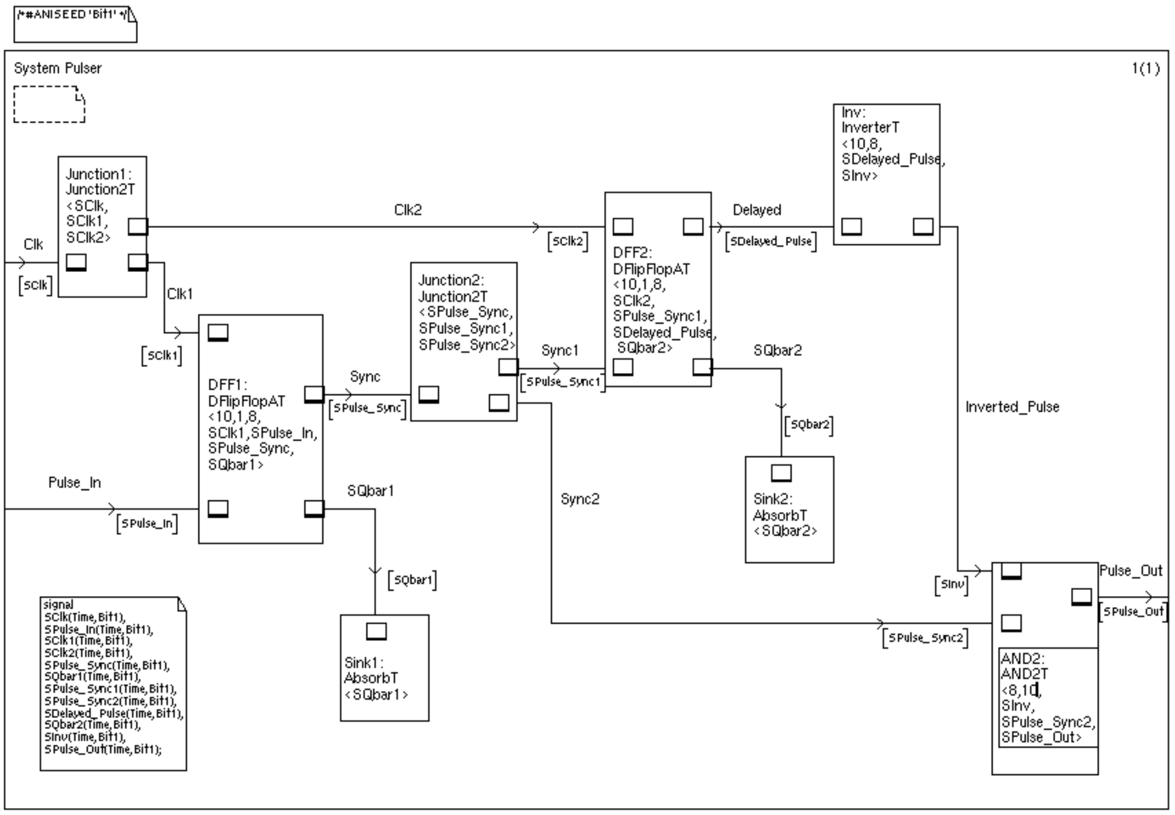


***Figure 48**. Example waveforms for the single-pulser (no delays considered)*

The SDL system constructed to simulate and analyse the single pulser is shown in figure 49. It consists of a system with eight blocks that are instances of block types contained in the library. In this figure the interconnecting wires for the circuit are modelled as SDL channels. They have been included for clarity but they are not compulsory.

DFF1 and DFF2 are instances of the D flip-flops with positive-edge triggering included in the library (*DFlipFlopAT*). The parameters given are the setup time (10 ns), holding time (1 ns) and the propagation delay (8 ns). These values have been selected as being representative of some D flip-flops available on the market. The signals that every flip-flop receives are also included as parameters. Gate names are not explicitly required in the diagram, but SDT still shows (figure 49) these small squares (within block bodies) where gate names are supposed to be. Adding gate names is time consuming and error prone, so they are inferred by *sdlctx,* the script that deals with context parameters as well.

The inverter and the two-input And gate are instantiated in a similar way. Parameters include the low-to-high and high-to-low propagation delays (values taken from an inverter 74H05 and a gate 74LS08) and the signals. All signals are declared in a text box on the left bottom corner of the figure.

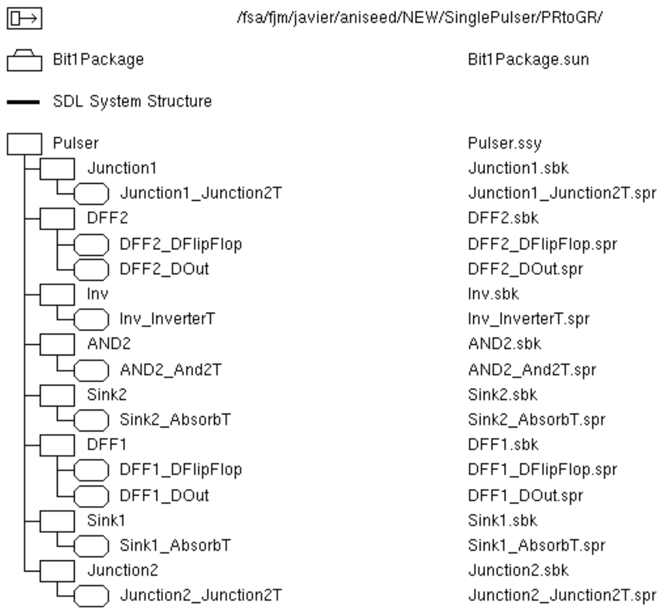

/*#ANISEED 'Bit1' */
System Pulser
1(1)
Junction1: Junction2T <SClk, SClk1, SClk2>
Clk
[SClk]
Clk2
[SClk2]
Clk1
[SClk1]
DFF1: DFlipFlopAT <10,1,8, SClk1,SPulse_In, SPulse_Sync, SQbar1>
Junction2: Junction2T <SPulse_Sync, SPulse_Sync1, SPulse_Sync2>
DFF2: DFlipFlopAT <10,1,8, SClk2, SPulse_Sync1, SDelayed_Pulse, SQbar2>
Delayed
[SDelayed_Pulse]
Inv: InverterT <10,8, SDelayed_Pulse, SInv>
Sync
[SPulse_Sync]
Sync1
[SPulse_Sync1]
SQbar2
[SQbar2]
Inverted_Pulse
Pulse_In
[SPulse_In]
SQbar1
[SQbar1]
Sync2
Sink2: AbsorbT <SQbar2>
Sink1: AbsorbT <SQbar1>
[SInv]
[SPulse_Sync2]
Pulse_Out
[SPulse_Out]
AND2: AND2T <8,10, SInv, SPulse_Sync2, SPulse_Out>

```
signal
SClk(Time,Bit1),
SPulse_In(Time,Bit1),
SClk1(Time,Bit1),
SClk2(Time,Bit1),
SPulse_Sync(Time,Bit1),
SQbar1(Time,Bit1),
SPulse_Sync1(Time,Bit1),
SPulse_Sync2(Time,Bit1),
SDelayed_Pulse(Time,Bit1),
SQbar2(Time,Bit1),
SInv(Time,Bit1),
SPulse_Out(Time,Bit1);
```

***Figure 49**. SDL implementation of the single pulser using the ANISEED library*

There are some other peculiarities in the SDL system given in figure 49. Electrical connections between wires are represented by instances of block type *Junction2T* (a timed junction with one input and two outputs). As described in chapter 3, a limitation of SDL is that an output cannot be broadcast to an arbitrary number of processes. To solve this problem, ANISEED uses junction components that model the connecting points of wires. In addition, process output signals have to be consumed even if not used. Unused terminals (such as the inverted outputs of the D flip-flops) are connected to instances of block type *AbsorbT.* The *Absorb* process type is ready to accept and absorb any signal. Both *Absorb* and *Junction* are part of the previous ANISEED library, but are also available in the new version.

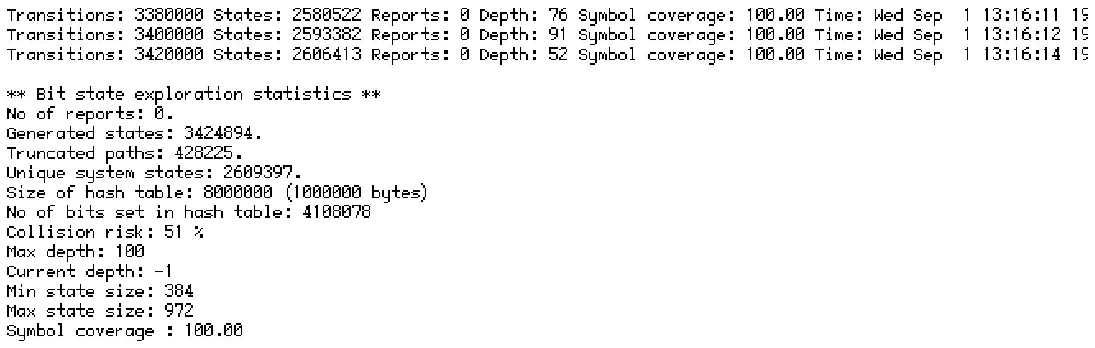
The single pulser specification was successfully validated using the SDT validator. To get round the compilation problems with the SDT code generator mentioned before, the PR file for the system was converted into GR format and imported into the tool. Figure 50 shows the structure of the re-constructed system. Files with extension "*.sbk*" are SDL blocks and files with extension "*.spr*" are processes. "*Pulser.ssy*" represents the whole system. Notice that the package *Bit1* is also included.



```
/fsa/fjm/javier/aniseed/NEW/SinglePulser/PRtoGR/
Bit1Package                        Bit1Package.sun
SDL System Structure
Pulser                             Pulser.ssy
  Junction1                        Junction1.sbk
    Junction1_Junction2T           Junction1_Junction2T.spr
  DFF2                             DFF2.sbk
    DFF2_DFlipFlop                 DFF2_DFlipFlop.spr
    DFF2_DOut                      DFF2_DOut.spr
  Inv                              Inv.sbk
    Inv_InverterT                  Inv_InverterT.spr
  AND2                             AND2.sbk
    AND2_And2T                     AND2_And2T.spr
  Sink2                            Sink2.sbk
    Sink2_AbsorbT                  Sink2_AbsorbT.spr
  DFF1                             DFF1.sbk
    DFF1_DFlipFlop                 DFF1_DFlipFlop.spr
    DFF1_DOut                      DFF1_DOut.spr
  Sink1                            Sink1.sbk
    Sink1_AbsorbT                  Sink1_AbsorbT.spr
  Junction2                        Junction2.sbk
    Junction2_Junction2T           Junction2_Junction2T.spr
```

Figure 50*. Structure of the SDL system for the single pulser*

Successful validation was achieved when using all but the exhaustive exploration algorithm. The bit state algorithm easily achieved 100% symbol coverage and no error reports were generated.

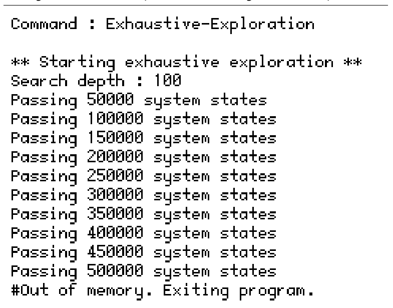


```
Transitions: 3380000 States: 2580522 Reports: 0 Depth: 76 Symbol coverage: 100.00 Time: Wed Sep  1 13:16:11 19
Transitions: 3400000 States: 2593382 Reports: 0 Depth: 91 Symbol coverage: 100.00 Time: Wed Sep  1 13:16:12 19
Transitions: 3420000 States: 2606413 Reports: 0 Depth: 52 Symbol coverage: 100.00 Time: Wed Sep  1 13:16:14 19

** Bit state exploration statistics **
No of reports: 0.
Generated states: 3424894.
Truncated paths: 428225.
Unique system states: 2609397.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 4108078
Collision risk: 51 %
Max depth: 100
Current depth: -1
Min state size: 384
Max state size: 972
Symbol coverage : 100.00
```

The power walk and random walk algorithms also finished without error reports. The resulting MSC traces given by the tool were analysed to check the functionality and behaviour of the system.

Exhaustive exploration failed after the computer running out of memory in just a few minutes. The machine used is a Sun workstation with 512 Mbytes of RAM memory, something that makes it difficult to understand how this algorithm can fail in such a short period of time. From previous discussions with Telelogic, it appears that SDT is severely limited (and not by RAM) in exhaustive exploration.



```
Command : Exhaustive-Exploration

** Starting exhaustive exploration **
Search depth : 100
Passing 50000 system states
Passing 100000 system states
Passing 150000 system states
Passing 200000 system states
Passing 250000 system states
Passing 300000 system states
Passing 350000 system states
Passing 400000 system states
Passing 450000 system states
Passing 500000 system states
#Out of memory. Exiting program.
```

After validation, the Single Pulser circuit was carefully simulated, using both the standard SDT simulator and the new run time library developed by Stephen Laing [31] for real-time hardware simulation with SDT. When appropriate signals were sent and all subsequent transitions executed before sending new signals, no difference in behaviour was noticed between these two simulator scheduling algorithms. The diagrams presented later in this chapter were obtained with the standard SDT simulator.

Some plans were made prior to the simulation, especially lists of the signals that had to be sent at certain times. The clock rate used to simulate the system was 10 MHz; that means 100 nanoseconds for the clock period. MSC charts were activated and the results were converted from MSC charts into timing diagrams (figure 51). Timing diagrams are frequently used in the analysis of electronic systems. These diagrams show various signals as a function of time. Several variables are usually plotted with the same time scale so that the times at which these variables change with respect to each other can easily be observed.

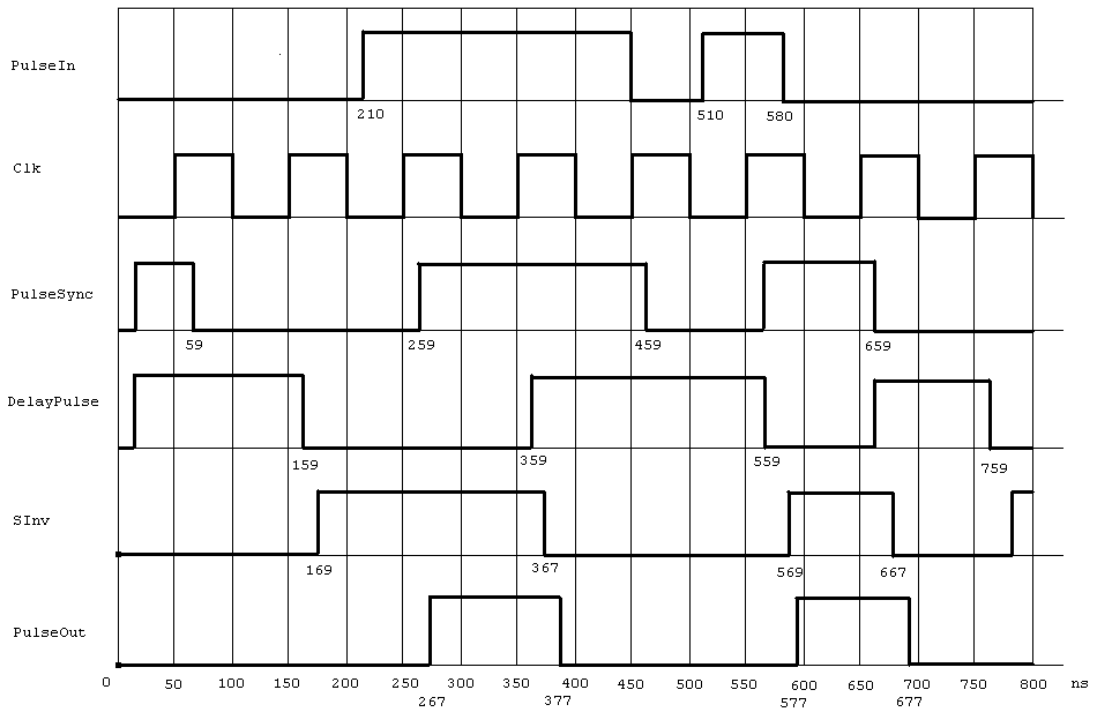


***Figure 51**. Timing diagram after simulating the SDL description of the single pulser*

The conclusion after simulation is that the single pulser circuit behaviour matches the specification and follows the properties stated in the benchmark document. Some interesting results were noticed. For example, when a *Pulse_In* signal arrives at the same instant as a Clock rising edge, no *Pulse_Out* is generated. The reason for this is the setup time of the flip-flop not being respected, a situation considered in our SDL specification (clock pulses during setup are simply ignored). The actual behaviour of a physical D flip-flop under these conditions is not specified by the manufacturer. Only normal operation is explained in datasheets, so non-determinism would apply to how a real flip-flop would behave in this case.

Before sending signals to the circuit it is necessary to execute all the initial transitions. All the elements in the circuit have initialisation sequences similar to the ones described in previous chapters. Outputs are randomly initialised and the circuit needs some time to stabilise. Deciding when the system is stable is not as easy as it might seem. In all time diagrams presented here, the duration of the first output pulse is longer than the clock period (110 ns instead of 100), something that goes against the specification of the circuit. The reason for this is the initial state of the second flip-flop and the inverter. In figure 52 the inverter output is high at time 150, when a positive edge clock transition is received and *PulseIn* is also at high level.

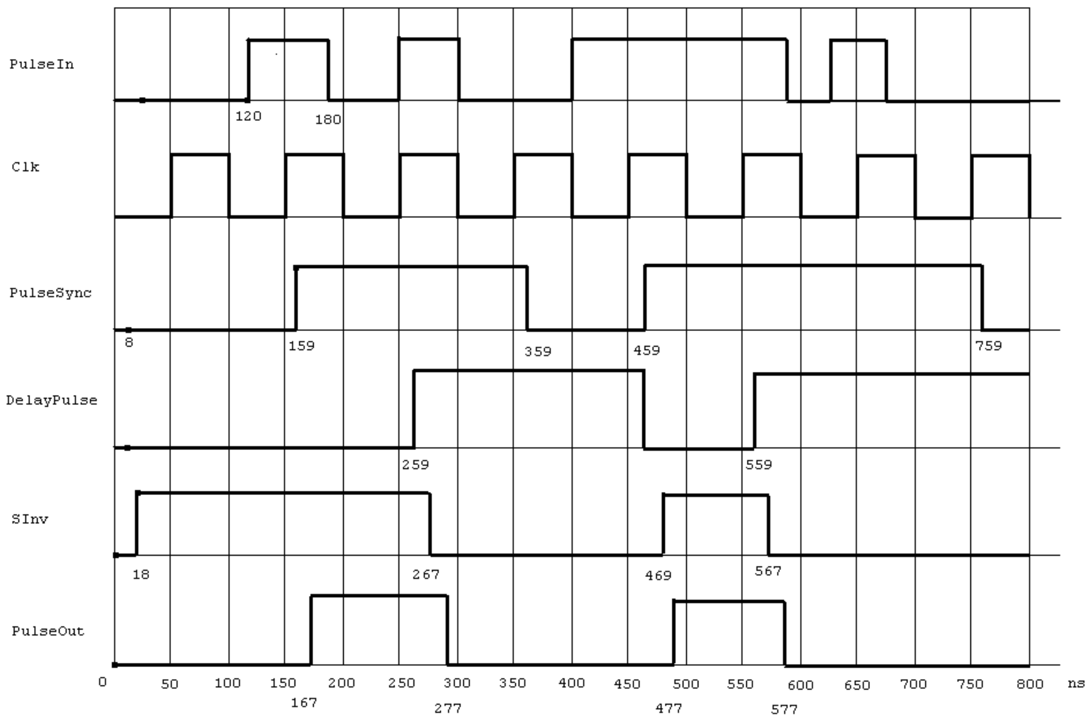


***Figure 52**. Timing diagram for the single pulser*

After 9 nanoseconds (1 ns for holding and 8 ns for propagation) the first flip-flop changes its output. Notice that the setup time is not part of the flip-flop response time here, as it had finished before the clock edge was received (setup finished at 120+10=130 ns). At time 159 the And gate has its both inputs high, so it starts a transition to set its output accordingly. After 8 ns (low-to-high propagation delay given as parameter to the gate) *PulseOut* goes to high level, unfortunately 10 ns before it should.

Figure 52 also shows that a transient variation in *PulseIn* between positive edges of the clock is irrelevant. Between times 550 and 650 *PulseIn* is low for some nanoseconds, so we might think that at time 650 a new output pulse sequence would be initiated (the previous output pulse has already finished). However, the first flip-flop only reads the state of the input when it receives positive clock edges, so it does not detect variations in input between clock active signals. After completing an output pulse the circuit needs some recovery time before a new one can be generated. *PulseIn* must be low during a period of time long enough to reach the next positive clock edge, so that the first flip-flop can effectively detect that *PulseIn* has been at low level.

Only the first output pulse has the problem commented above. As shown in figures 51 and 52, the second pulse is right, since it lasts exactly one clock period (100 ns). Figure 53 shows a sequence of several consecutive output pulses. All but the first are correct. In the second output pulse, for example, the clock edge is received at time 250. Nine nanoseconds later the first flip-flop changes its state, but now the second flip-flop changes too, since its input *Pulse_sync* was zero. The inverter now needs 10 nanoseconds to set its output high. In the meantime, the AND gate has one of its inputs at low level, so it must wait until the inverter has finished its propagation time. For this reason, the output pulse is 10 nanosecond shorter that the first one. As shown in figure 53, all pulses but the very first one are correct.

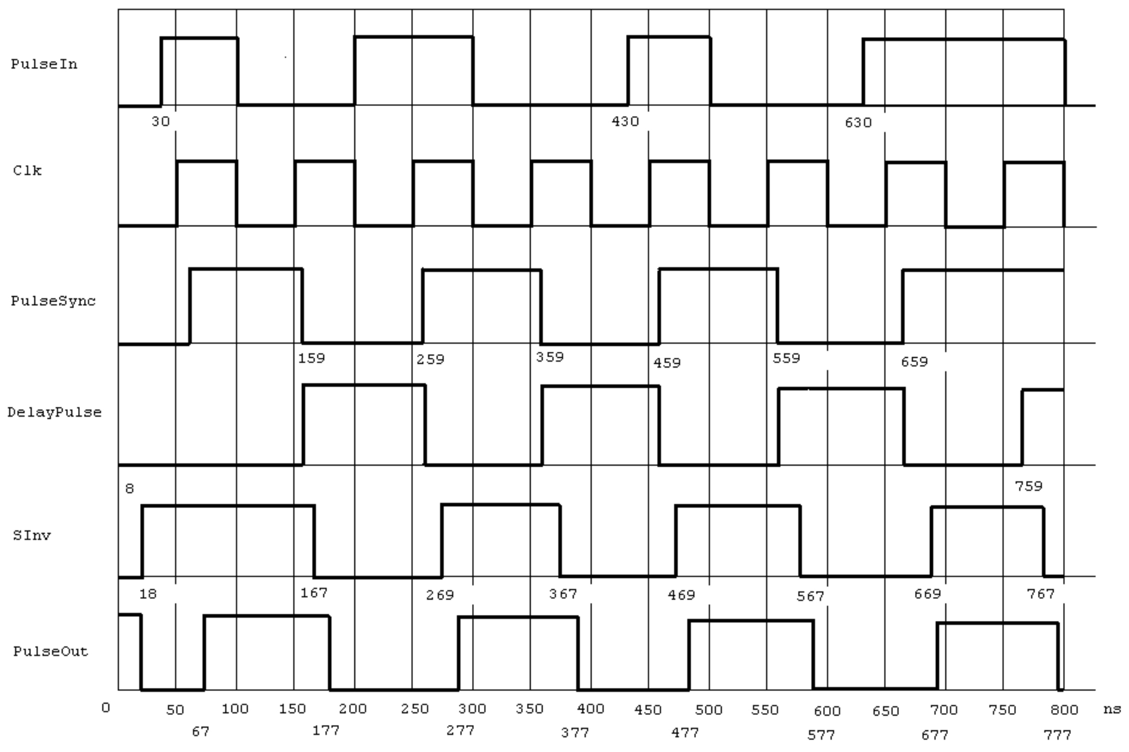


Figure 53. *Time diagram showing a consecutive sequence of output pulses*

8 Conclusions

SDL is a well-structured and user-friendly language. The language itself combined with the facilities included in Telelogic SDT made this work a rewarding experience. However, even with the help of SDL and SDT, formally specifying of hardware components is not an easy task.

Describing truth tables and the basic behaviour of electronic components is usually a rather straightforward activity. However, timing constraints are always the trickiest aspect of the specification. Dealing with timers, propagation delays and transient conditions have proved the hardest part of any specification.

After achieving a clear and precise SDL specification for a device's normal operation, there was usually a need for further questions. Datasheets describe the behaviour of components under certain assumptions. If the 'rules of the game' are not respected, electronic components will probably behave in a non-deterministic way. Something as simple as ignoring any strange conditions in the specification is actually an explicit decision. More often than desired these reflections lead to very intricate thoughts about the best way of dealing with transient or abnormal behaviour while keeping the specifications simple.

Validating systems with the SDT validator is more a question of time than skill. The tool really helps in finding the most bizarre combinations of inputs, outputs and adverse circumstances that can make everything go wrong. The bad news about this approach is that, even in a small size description, thousands of possible combinations may exist. Long hours of careful analysis of Message Sequence Charts are then the central part of the job. Besides automatically generated MSCs, some SDT features such as the navigator, the watch window to trace variables, and the coverage viewer to detect parts not fully explored have proved really useful.

SDT is quite an impressive tool. However, it creates real problems sometimes. More than once the graphical editor was suddenly closed, a big core dump was created and SDT closed without any chance of getting the work saved. Some care had also to be taken about "dead" processes running out of control in the background after closing the tool.

SDL and the ANISEED approach have been shown to be very much applicable to the realm of hardware analysis and design. It has been possible to formally specify a whole range of electronic components that now can be used to create and analyse more complex devices and electronic systems. The deep software roots of SDL make it clear that software-hardware co-design can also be achieved.

There is a great deal of scope for future work on hardware description in SDL and ANISEED in particular. These are some of the future options:

- Providing a nicer graphical front-end that allows entry of circuit diagrams more closely resembling those an engineer would draw.
- Further extension of the library.
- Investigating exhaustive validation further. The SDT validator fails very rapidly, even in machines with a considerable amount of RAM.
- Carrying out new case studies.
- Automatic generation of tests from descriptions.
- Carrying out a practical comparison of ANISEED and other traditional approaches such as VHDL, for example.
- Looking at hardware-software co-design with SDL.

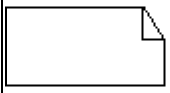
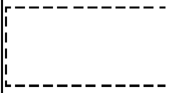
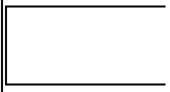
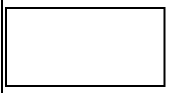
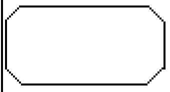
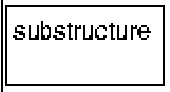
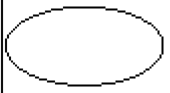
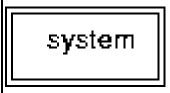
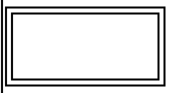
References

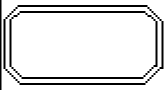
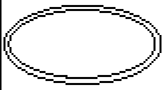
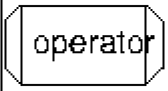
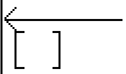
[1] ITU-T. *“Specification and Description Language (SDL)”*. Recommendation Z.100. International Telecommunications Union, Geneva, 1996.
[2] C. H. Roth. *“Fundamentals of logic design”*. West publishing company. 1992.
[3] IEEE. *“VHSIC Hardware Design Language”*. IEEE 1076. Institution of Electrical and Electronic Engineers Press, New York, USA, 1992.
[4] IEEE. *“IEEE Standard Hardware Design Language based on the Verilog Hardware Design Language”*. IEEE 1364. Institution of Electrical and Electronic Engineers Press, New York, USA, 1995.
[5] A. Janstch, S Kumar et al. *“Comparison of six languages for system level descriptions of telecom systems”*. Royal Institute of Technology. Stockholm, Sweden. 1997.
[6] J. Armstrong, B. Däcker et al. *“Erlang white paper”*. http://www.erlang.org
[7] S. P. Jones and J. Hughes editors. *“Haskell 98 Report”*. http://haskell.org
[8] J. M. Daveau, G. Fernandes et al. *“VHDL generation from SDL specifications”*. CHDL 97, April, 1997.
[9] I. S. Bonatti and R.J. Figuerido. *“An algorithm for the translation of SDL into synthesizable VHDL”*. Current Issues in Electronic Modeling, Vol. 3, August 1995.
[10] O. Pulkkienen and K. Kronlof. *“Integration of SDL and VHDL for High Level Digital design”*. Proceedings of the European Design Automation Conference with Euro-VHDL, September 1992, pp 624-629.
[11] B. Lutter, W. Glunz and F. J. Ramming. *“Using VHDL for simulation of SDL Specifications”*. Proceedings of EURO-VHDL 1992, pages 630-635.
[12] T. Ben Ismail, M. Abid, K. O’Brien, A. A. Jerraya. *“An Approach for Hardware-Software Co-design”*, RSP’94, Grenoble, France, June 1994.
[13] W. M. Loucks, B. J. Doray, D. G. Agnew. *“Experiences In Real Time Hardware-Software Co-simulation”*. Proc VHDL Int. Users Forum (VIUF), Ottawa, Canada, pp. 47-57, April 1993.
[14] B. Svantesson, S. Kumar and A. Hemani. “A methodology and algorithms for efficient inter-process communication synthesis from system descriptions in SDL”.
[15] ISO. Information Processing. Open systems Interconnection. *“LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour”*. ISO/IEC 8807. International Organization for Standardization, Geneva, 1989.
[16] K .J. Turner and R. 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.
[17] Ji He and K. J. Turner. *“Extended DILL: Digital logic with LOTOS”*. Technical Report CSM-142, Department of Computing Science and Mathematics, University of Stirling, UK, Nov. 1997.
[18] Ji He and K. J. Turner. *“Timed DILL: Digital logic with LOTOS”*. Technical Report CSM-145, Department of Computing Science and Mathematics, University of Stirling, UK, Apr. 1998.
[19] ITU-T. *“Message Sequence Chart (MSC)”*. Recommendation Z.120. September 1994.
[20] 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.
[21] Telelogic AB. SDT 3.1: *“Tutorial on SDT Tools”*. Malmø, Sweden, 1996.
[22] K. J. Turner, G. Csopaki and S. D. Laing. *“Hardware Timing Analysis with SDL”*. January 1999.
[23] J.-M. Daveau, G. F. Marchioro, T. Ben Ismail, and A. A. Jerraya. *”COSMOS: An SDL based hardware/software co-design environment”*. Current Issues In Electronic Modeling, 8:59–88, 1997.
[24] T. Hadlich and T. Szczepanski. *“The ODE system – An SDL-based approach to hardware-software co-design”*. In C. Muller-Schlor, F. Geerinckx, B. Stanford-Smith, and R. van Riet, editors. Embedded Microprocessor systems, pages 269-281. IOS Press, Amsterdam, Netherlands, 1996.
[25] Telelogic. *TAU 3.5 Manuals*. Telelogic, Malmø, Sweden, May. 1999.
[26] Anders Ek. “Automatic Debugging of Communicating Systems using the SDT Validator”. Telelogic AB, Malmø, Sweden. 1997.
[27] R. Seindal. *“GNU m4 version 1.4”*. Technical report, Free Software Foundation, 1997.
[28] J. Staunstrup and T. Kropf. *“IFIP WG 10.5 benchmark circuits v1.2.0”*. http://goethe.ira.uka.de/hvg/benchmarks.html

[29] S. D. Johnsons, P. S. Miner, and A. Camilleri. *"Studies of the single pulser in various reasoning systems"*. In T. Kropf and R. Kumar editors. Proc. 2nd International Conference on Theorem Provers in Circuit Design (TPCD94), volume 901 of Lecture Notes in Computer Science, pages 126–145, Bad Herrenalb, Germany, September 1994. Springer-Verlag. published 1995.
[30] D. Winkel and F. Prosser. *"The art of digital design"*. Prentice Hall Inc, 1980.
[31] S. Laing. *"Hardware Specification and Analysis in SDL"*. Honours Dissertation. Department of Computing Science and Mathematics. University of Stirling. April 1999.

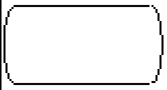
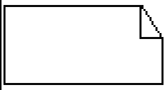
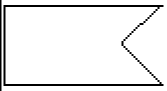
A. SDL Notation

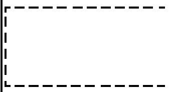
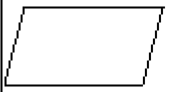
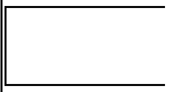
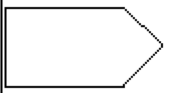
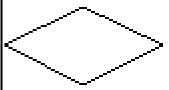
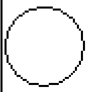
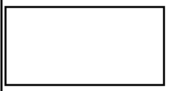
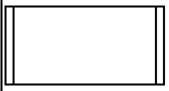
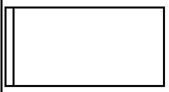
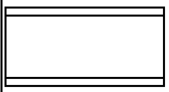
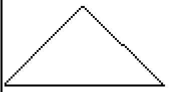
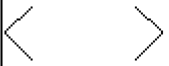
Symbols on Interaction Pages

Symbol Appearance	Symbol Name	References to Z.100
	Text	z100: 2.5.4 Signal z100: 2.5.5 Signal list z100: 5.2.1 Newtype z100: 5.3.1.9 Syntype z100: 5.3.1.13 Synonym z100: 5.3.1.12.1 Generator z100: 4.13 Remote variable z100: 4.14 Remote procedure
	Comment	z100: 2.2.6
	Text extension	z100: 2.2.7 (depends on the symbol connected to)
	Block reference	z100: 2.4.2 Block definition z100: 6.1.3.2 Block def based on block type z100: 6.1.2 Type expression z100: 6.2 Actual context parameters
	Process reference	z100: 2.4.3 Process definition z100: 2.4.4 Number of instances z100: 6.1.3.3 Process def based on block type z100: 6.1.2 Type expression z100: 6.2 Actual context parameters
 substructure	Block substructure reference	z100: 3.2.2
	Service reference	z100: 2.4.4
 system	System type	z100: 6.1.1.1
	Block type	z100: 6.1.1.2

	Process type	Z100: 6.1.1.3
	Service type	Z100: 6.1.1.4
	Operator reference (an SDL-GR extension defined in SDT)	Z100: 5.3.2 Referenced operator in SDL-PR
	Gate	Z100: 6.1.4 Gate Z100: 2.5.5 Signal list

Symbols on Flow Pages

Symbol Appearance	Symbol Name	References to Z.100
	State or nextstate	Z100: 2.6.3 State Z100: 4.4 Asterisk state Z100: 4.5 Multiple appearence of state Z100: 2.6.8.2.1 Nextstate Z100: 4.9 Dash nextstate
	Text	Z100: 2.5.4 Signal Z100: 2.5.5 Signal list Z100: 5.2.1 Newtype Z100: 5.3.1.9 Syntype Z100: 5.3.1.13 Synonym Z100: 5.3.1.12.1 Generator Z100: 2.6.1.1 Variable Z100: 2.6.1.2 View Z100: 2.8 Timer Z100: 4.13 Remote variable Z100: 4.13 Imported variable Z100: 4.14 Remote procedure Z100: 4.14 Imported procedure
	Input	Z100: 2.6.4 Input Z100: 4.6 Asterisk input Z100: 4.14 Remote procedure input Z100: 6.3.3 Virtual transition z100: 5.4.3 Variable z100: 5.4.3.1 Indexed variable z100: 5.4.3.2 Field Variable

	Comment	Z100: 2.2.6
	Save	Z100: 2.6.5 Save Z100: 4.7 Asterisk save Z100: 6.3.3 Virtual save
	Text extension	Z100: 2.2.7 (depends on the symbol connected to)
	Output	Z100: 2.7.4
	Decision	Z100: 2.7.5 Decision Z100: 5.3.1.9.1 Range condition Z100: 2.2.3 Informal text
	In-connector or out-connector	Z100: 2.6.7 In-connector Z100: 2.6.8.2.2 Out-connector
	Task, set, reset or export	Z100: 2.7.1 Task Z100: 5.4.3 Assignment Z100: 2.8 Set, Reset Z100: 4.13 Export
	Procedure call	Z100: 2.7.3 Call Z100: 2.7.2 Actual parameters
	Macro call	Z100: 4.2.3
	Create request	Z100: 2.7.2
	Transition option	Z100: 4.3.4 Transition option Z100: 5.3.1.9.1 Range condition
	Continuous signal or enabling condition	Z100: 4.11 Continuous signal Z100: 4.12 Enabling condition

B. List of New Library Components

Naming conventions for logic components are as follows:

decoder	<func>Decoder[<abst>][<time>][<mult>] e.g. BcdDecDecoderAT, TwoFourDecoderT
encoder	<func>Encoder[<abst>][<time>][<mult>] e.g. EightThreeEncoder
flip-flop	<func>FlipFlop[<abst>][<time>][<trig][<pre>] e.g. DFlipFlopA, JKFlipFlopATN, RSFlipFlopATNP, TFlipFlopAT
junction	Junction<outputs>[<time>][<mult>] e.g. Junction2T, Junction4TM
merge	Merge<outputs>[<time>][<mult>] e.g. Merge2TM, MergeT
n-ary logic	<func><inputs>[<time>][<mult>][<tris>] e.g. And2T, Xor4TM, Nor8MH
nullary	<func>[<time>][<mult>] e.g. AbsorbT, OneTM, Zero
split	Split<outputs>[<time>][<mult>] e.g. Split2TM, SplitH, SplitTL
unary	<func>[<time>][<mult>] e.g. InverterT, RepeaterM, RepeaterTM

variants	<abst>	A/I	(abstract/intermediate, default gate-level)
	<func>		(function)
	<mult>	M	(multi-bit, default single-bit)
	<pre>	P	(preset/clear, default neither)
	<time>	T	(timed, default untimed)
	<trig>	N	(negative FF trigger, default positive)
	<tris>	H/L	(high/low tristate enable, default neither)

Naming conventions for context parameters and formal parameters are as follows:

data values	BIp[<number>] B[Next]Op B<func>[Next] e.g. BIp1, BIp0, BNextOp, BNextQ
input gates	Ip[<number>] <func> e.g C, D, Ip, Ip1
input signals	S[<func>][Ip][<number>](<parameters>) e.g. SDIp (Time, BitM), SIp (Bit1), SIp0 (Time, Bit1)

output gates	Op[<number>]
	<func>
	e.g Op, Op2, Q, QBar
output signals	S[<func>][Op][<number>](<parameters>)
	e.g. SOp (Bit1), SOp0 (Time, Bit1), SQ (Time, Bit1)
parameter values	TDelay<func>
	e.g. TDelay1, TDelaySetup
timers	T<func>
	e.g. T1, THold
times	TIp, Top

Aniseed_Coder (all components timed)

EightThreeEncoderAT	Eight to three lines priority encoder
BcdDecDecoderAT	BCD to decimal decoder
TwoFourDecoderAT	Two to four lines decoder

Aniseed_FlipFlop

DFlipFlopAT	D flip-flop positive edge triggering
DFlipFlopATN	D flip-flop negative edge triggering
DFlipFlopATP	D flip-flop positive edge triggering with preset and clear
DFlipFlopATNP	D flip-flop negative edge triggering with preset and clear
MSDFlipFlopATP	Master–slave D flip-flop with preset and clear
JKFlipFlopAT	JK flip-flop positive edge triggering
JKFlipFlopATN	JK flip-flop negative edge triggering
JKFlipFlopATP	JK flip-flop positive edge triggering with preset and clear
JKFlipFlopATNP	JK flip-flop negative edge triggering with preset and clear
MSJKFlipFlopATP	Master-slave JK flip-flop with preset and clear
RSFlipFlopAT	RS flip-flop positive edge triggering
RSFlipFlopATN	RS flip-flop negative edge triggering
RSFlipFlopATP	RS flip-flop positive edge triggering with preset and clear
RSFlipFlopATNP	RS flip-flop negative edge triggering with preset and clear
MSRSFlipFlopATP	Master-slave RS flip-flop with preset and clear
TFlipFlopAT	T flip-flop positive edge triggering
TFlipFlopATN	T flip-flop negative edge triggering
MSTFlipFlopAT	Master-slave T flip-flop
TFlipFlopATP	T flip-flop positive edge triggering with preset and clear
TFlipFlopATNP	T flip-flop negative edge triggering with preset and clear

Aniseed_Mux (all components timed)

FourOneMultiplexerAT	Four to one line multiplexer
TwoFourDemultiplexerAT	Two to four line demultiplexer

Aniseed_Trigate (all components tri-state)

InverterH	Untimed single-bit inverter with high-level enable
InverterTH	Timed single-bit inverter with high-level enable
InverterMH	Untimed multi-bit inverter with high-level enable
InverterTMH	Timed multi-bit inverter with high-level enable
InverterL	untimed single-bit inverter with low-level enable
InverterTL	timed single-bit inverter with low-level enable
InverterML	untimed multi-bit inverter with low-level enable
InverterTML	timed multi-bit inverter with low-level enable
And2H	untimed single-bit “and” gate with 2 inputs with high-level enable
And3H	untimed single-bit “and” gate with 3 inputs with high-level enable
And4H	untimed single-bit “and” gate with 4 inputs with high-level enable
And8H	untimed single-bit “and” gate with 8 inputs with high-level enable
And2L	untimed single-bit “and” gate with 2 inputs with low-level enable
And3L	untimed single-bit “and” gate with 3 inputs with low-level enable
And4L	untimed single-bit “and” gate with 4 inputs with low-level enable
And8L	untimed single-bit “and” gate with 8 inputs with low-level enable
And2TH	timed single-bit “and” gate with 2 inputs with high-level enable
And3TH	timed single-bit “and” gate with 3 inputs with high-level enable
And4TH	timed single-bit “and” gate with 4 inputs with high-level enable
And8TH	timed single-bit “and” gate with 8 inputs with high-level enable
And2TL	timed single-bit “and” gate with 2 inputs with low-level enable
And3TL	timed single-bit “and” gate with 3 inputs with low-level enable
And4TL	timed single-bit “and” gate with 4 inputs with low-level enable
And8TL	timed single-bit “and” gate with 8 inputs with low-level enable
And2MH	untimed multi-bit “and” gate with 2 inputs with high-level enable
And3MH	untimed multi-bit “and” gate with 3 inputs with high-level enable
And4MH	untimed multi-bit “and” gate with 4 inputs with high-level enable
And8MH	untimed multi-bit “and” gate with 8 inputs with high-level enable
And2ML	untimed multi-bit “and” gate with 2 inputs with low-level enable
And3ML	untimed multi-bit “and” gate with 3 inputs with low-level enable
And4ML	untimed multi-bit “and” gate with 4 inputs with low-level enable
And8ML	untimed multi-bit “and” gate with 8 inputs with low-level enable
And2TMH	timed multi-bit “and” gate with 2 inputs with high-level enable
And3TMH	timed multi-bit “and” gate with 3 inputs with high-level enable
And4TMH	timed multi-bit “and” gate with 4 inputs with high-level enable
And8TMH	timed multi-bit “and” gate with 8 inputs with high-level enable
And2TML	timed multi-bit “and” gate with 2 inputs with low-level enable
And3TML	timed multi-bit “and” gate with 3 inputs with low-level enable
And4TML	timed multi-bit “and” gate with 4 inputs with low-level enable
And8TML	timed multi-bit “and” gate with 8 inputs with low-level enable
Nand2H	untimed single-bit “nand” gate with 2 inputs with high-level enable
Nand3H	untimed single-bit “nand” gate with 3 inputs with high-level enable
Nand4H	untimed single-bit “nand” gate with 4 inputs with high-level enable
Nand8H	untimed single-bit “nand” gate with 8 inputs with low-level enable
Nand2L	untimed single-bit “nand” gate with 2 inputs with low-level enable
Nand3L	untimed single-bit “nand” gate with 3 inputs with low-level enable
Nand4L	untimed single-bit “nand” gate with 4 inputs with low-level enable
Nand8L	untimed single-bit “nand” gate with 8 inputs with low-level enable
Nand2TH	timed single-bit “nand” gate with 2 inputs with high-level enable
Nand3TH	timed single-bit “nand” gate with 3 inputs with high-level enable
Nand4TH	timed single-bit “nand” gate with 4 inputs with high-level enable

Nand8TH	timed single-bit “nand” gate with 8 inputs with high-level enable
Nand2TL	timed single-bit “nand” gate with 2 inputs with low-level enable
Nand3TL	timed single-bit “nand” gate with 3 inputs with low-level enable
Nand4TL	timed single-bit “nand” gate with 4 inputs with low-level enable
Nand8TL	timed single-bit “nand” gate with 8 inputs with low-level enable
Nand2MH	untimed multi-bit “nand” gate with 2 inputs with high-level enable
Nand3MH	untimed multi-bit “nand” gate with 3 inputs with high-level enable
Nand4MH	untimed multi-bit “nand” gate with 4 inputs with high-level enable
Nand8MH	untimed multi-bit “nand” gate with 8 inputs with high-level enable
Nand2ML	untimed multi-bit “nand” gate with 2 inputs with low-level enable
Nand3ML	untimed multi-bit “nand” gate with 3 inputs with low-level enable
Nand4ML	untimed multi-bit “nand” gate with 4 inputs with low-level enable
Nand8ML	untimed multi-bit “nand” gate with 8 inputs with low-level enable
Nand2TMH	timed multi-bit “nand” gate with 2 inputs with high-level enable
Nand3TMH	timed multi-bit “nand” gate with 3 inputs with high-level enable
Nand4TMH	timed multi-bit “nand” gate with 4 inputs with high-level enable
Nand8TMH	timed multi-bit “nand” gate with 8 inputs with high-level enable
Nand2TML	timed multi-bit “nand” gate with 2 inputs with low-level enable
Nand3TML	timed multi-bit “nand” gate with 3 inputs with low-level enable
Nand4TML	timed multi-bit “nand” gate with 4 inputs with low-level enable
Nand8TML	timed multi-bit “nand” gate with 8 inputs with low-level enable
Or2H	untimed single-bit “or” gate with 2 inputs with high-level enable
Or3H	untimed single-bit “or” gate with 3 inputs with high-level enable
Or4H	untimed single-bit “or” gate with 4 inputs with high-level enable
Or8H	untimed single-bit “or” gate with 8 inputs with low-level enable
Or2L	untimed single-bit “or” gate with 2 inputs with low-level enable
Or3L	untimed single-bit “or” gate with 3 inputs with low-level enable
Or4L	untimed single-bit “or” gate with 4 inputs with low-level enable
Or8L	untimed single-bit “or” gate with 8 inputs with low-level enable
Or2TH	timed single-bit “or” gate with 2 inputs with high-level enable
Or3TH	timed single-bit “or” gate with 3 inputs with high-level enable
Or4TH	timed single-bit “or” gate with 4 inputs with high-level enable
Or8TH	timed single-bit “or” gate with 8 inputs with high-level enable
Or2TL	timed single-bit “or” gate with 2 inputs with low-level enable
Or3TL	timed single-bit “or” gate with 3 inputs with low-level enable
Or4TL	timed single-bit “or” gate with 4 inputs with low-level enable
Or8TL	timed single-bit “or” gate with 8 inputs with low-level enable
Or2MH	untimed multi-bit “or” gate with 2 inputs with high-level enable
Or3MH	untimed multi-bit “or” gate with 3 inputs with high-level enable
Or4MH	untimed multi-bit “or” gate with 4 inputs with high-level enable
Or8MH	untimed multi-bit “or” gate with 8 inputs with high-level enable
Or2ML	untimed multi-bit “or” gate with 2 inputs with low-level enable
Or3ML	untimed multi-bit “or” gate with 3 inputs with low-level enable
Or4ML	untimed multi-bit “or” gate with 4 inputs with low-level enable
Or8ML	untimed multi-bit “or” gate with 8 inputs with low-level enable
Or2TMH	timed multi-bit “or” gate with 2 inputs with high-level enable
Or3TMH	timed multi-bit “or” gate with 3 inputs with high-level enable
Or4TMH	timed multi-bit “or” gate with 4 inputs with high-level enable
Or8TMH	timed multi-bit “or” gate with 8 inputs with high-level enable
Or2TML	timed multi-bit “or” gate with 2 inputs with low-level enable
Or3TML	timed multi-bit “or” gate with 3 inputs with low-level enable
Or4TML	timed multi-bit “or” gate with 4 inputs with low-level enable
Or8TML	timed multi-bit “or” gate with 8 inputs with low-level enable

Nor2H	untimed single-bit “nor” gate with 2 inputs with high-level enable
Nor3H	untimed single-bit “nor” gate with 3 inputs with high-level enable
Nor4H	untimed single-bit “nor” gate with 4 inputs with high-level enable
Nor8H	untimed single-bit “nor” gate with 8 inputs with low-level enable
Nor2L	untimed single-bit “nor” gate with 2 inputs with low-level enable
Nor3L	untimed single-bit “nor” gate with 3 inputs with low-level enable
Nor4L	untimed single-bit “nor” gate with 4 inputs with low-level enable
Nor8L	untimed single-bit “nor” gate with 8 inputs with low-level enable
Nor2TH	timed single-bit “nor” gate with 2 inputs with high-level enable
Nor3TH	timed single-bit “nor” gate with 3 inputs with high-level enable
Nor4TH	timed single-bit “nor” gate with 4 inputs with high-level enable
Nor8TH	timed single-bit “nor” gate with 8 inputs with high-level enable
Nor2TL	timed single-bit “nor” gate with 2 inputs with low-level enable
Nor3TL	timed single-bit “nor” gate with 3 inputs with low-level enable
Nor4TL	timed single-bit “nor” gate with 4 inputs with low-level enable
Nor8TL	timed single-bit “nor” gate with 8 inputs with low-level enable
Nor2MH	untimed multi-bit “nor” gate with 2 inputs with high-level enable
Nor3MH	untimed multi-bit “nor” gate with 3 inputs with high-level enable
Nor4MH	untimed multi-bit “nor” gate with 4 inputs with high-level enable
Nor8MH	untimed multi-bit “nor” gate with 8 inputs with high-level enable
Nor2ML	untimed multi-bit “nor” gate with 2 inputs with low-level enable
Nor3ML	untimed multi-bit “nor” gate with 3 inputs with low-level enable
Nor4ML	untimed multi-bit “nor” gate with 4 inputs with low-level enable
Nor8ML	untimed multi-bit “nor” gate with 8 inputs with low-level enable
Nor2TMH	timed multi-bit “nor” gate with 2 inputs with high-level enable
Nor3TMH	timed multi-bit “nor” gate with 3 inputs with high-level enable
Nor4TMH	timed multi-bit “nor” gate with 4 inputs with high-level enable
Nor8TMH	timed multi-bit “nor” gate with 8 inputs with high-level enable
Nor2TML	Timed multi-bit “nor” gate with 2 inputs with low-level enable
Nor3TML	Timed multi-bit “nor” gate with 3 inputs with low-level enable
Nor4TML	Timed multi-bit “nor” gate with 4 inputs with low-level enable
Nor8TML	Timed multi-bit “nor” gate with 8 inputs with low-level enable
Xor2H	Untimed single-bit “xor” gate with 2 inputs with high-level enable
Xor3H	Untimed single-bit “xor” gate with 3 inputs with high-level enable
Xor4H	Untimed single-bit “xor” gate with 4 inputs with high-level enable
Xor8H	Untimed single-bit “xor” gate with 8 inputs with low-level enable
Xor2L	Untimed single-bit “xor” gate with 2 inputs with low-level enable
Xor3L	Untimed single-bit “xor” gate with 3 inputs with low-level enable
Xor4L	Untimed single-bit “xor” gate with 4 inputs with low-level enable
Xor8L	Untimed single-bit “xor” gate with 8 inputs with low-level enable
Xor2TH	Timed single-bit “xor” gate with 2 inputs with high-level enable
Xor3TH	Timed single-bit “xor” gate with 3 inputs with high-level enable
Xor4TH	Timed single-bit “xor” gate with 4 inputs with high-level enable
Xor8TH	Timed single-bit “xor” gate with 8 inputs with high-level enable
Xor2TL	Timed single-bit “xor” gate with 2 inputs with low-level enable
Xor3TL	Timed single-bit “xor” gate with 3 inputs with low-level enable
Xor4TL	Timed single-bit “xor” gate with 4 inputs with low-level enable
Xor8TL	Timed single-bit “xor” gate with 8 inputs with low-level enable
Xor2MH	Untimed multi-bit “xor” gate with 2 inputs with high-level enable
Xor3MH	Untimed multi-bit “xor” gate with 3 inputs with high-level enable
Xor4MH	Untimed multi-bit “xor” gate with 4 inputs with high-level enable
Xor8MH	Untimed multi-bit “xor” gate with 8 inputs with high-level enable
Xor2ML	Untimed multi-bit “xor” gate with 2 inputs with low-level enable

Xor3ML	Untimed multi-bit “xor” gate with 3 inputs with low-level enable
Xor4ML	Untimed multi-bit “xor” gate with 4 inputs with low-level enable
Xor8ML	Untimed multi-bit “xor” gate with 8 inputs with low-level enable
Xor2TMH	Timed multi-bit “xor” gate with 2 inputs with high-level enable
Xor3TMH	Timed multi-bit “xor” gate with 3 inputs with high-level enable
Xor4TMH	Timed multi-bit “xor” gate with 4 inputs with high-level enable
Xor8TMH	Timed multi-bit “xor” gate with 8 inputs with high-level enable
Xor2TML	timed multi-bit “xor” gate with 2 inputs with low-level enable
Xor3TML	timed multi-bit “xor” gate with 3 inputs with low-level enable
Xor4TML	timed multi-bit “xor” gate with 4 inputs with low-level enable
Xor8TML	timed multi-bit “xor” gate with 8 inputs with low-level enable
Xnor2H	untimed single-bit “xnor” gate with 2 inputs with high-level enable
Xnor3H	untimed single-bit “xnor” gate with 3 inputs with high-level enable
Xnor4H	untimed single-bit “xnor” gate with 4 inputs with high-level enable
Xnor8H	untimed single-bit “xnor” gate with 8 inputs with low-level enable
Xnor2L	untimed single-bit “xnor” gate with 2 inputs with low-level enable
Xnor3L	untimed single-bit “xnor” gate with 3 inputs with low-level enable
Xnor4L	untimed single-bit “xnor” gate with 4 inputs with low-level enable
Xnor8L	untimed single-bit “xnor” gate with 8 inputs with low-level enable
Xnor2TH	timed single-bit “xnor” gate with 2 inputs with high-level enable
Xnor3TH	timed single-bit “xnor” gate with 3 inputs with high-level enable
Xnor4TH	timed single-bit “xnor” gate with 4 inputs with high-level enable
Xnor8TH	timed single-bit “xnor” gate with 8 inputs with high-level enable
Xnor2TL	timed single-bit “xnor” gate with 2 inputs with low-level enable
Xnor3TL	timed single-bit “xnor” gate with 3 inputs with low-level enable
Xnor4TL	timed single-bit “xnor” gate with 4 inputs with low-level enable
Xno8TL	timed single-bit “xnor” gate with 8 inputs with low-level enable
Xnor2MH	untimed multi-bit “xnor” gate with 2 inputs with high-level enable
Xnor3MH	untimed multi-bit “xnor” gate with 3 inputs with high-level enable
Xnor4MH	untimed multi-bit “xnor” gate with 4 inputs with high-level enable
Xnor8MH	untimed multi-bit “xnor” gate with 8 inputs with high-level enable
Xnor2ML	untimed multi-bit “xnor” gate with 2 inputs with low-level enable
Xnor3ML	untimed multi-bit “xnor” gate with 3 inputs with low-level enable
Xnor4ML	untimed multi-bit “xnor” gate with 4 inputs with low-level enable
Xnor8ML	untimed multi-bit “xnor” gate with 8 inputs with low-level enable
Xnor2TMH	timed multi-bit “xnor” gate with 2 inputs with high-level enable
Xnor3TMH	timed multi-bit “xnor” gate with 3 inputs with high-level enable
Xnor4TMH	timed multi-bit “xnor” gate with 4 inputs with high-level enable
Xnor8TMH	timed multi-bit “xnor” gate with 8 inputs with high-level enable
Xnor2TML	timed multi-bit “xnor” gate with 2 inputs with low-level enable
Xnor3TML	timed multi-bit “xnor” gate with 3 inputs with low-level enable
Xnor4TML	timed multi-bit “xnor” gate with 4 inputs with low-level enable
Xnor8TML	timed multi-bit “xnor” gate with 8 inputs with low-level enable