High-level modelling languages

J. Evans
CERN, Geneva, Switzerland

Abstract
This paper gives an introduction to the latest developments in modern electronic design methodology. It will give a brief history of the evolution of design software in an attempt to explain the seemingly haphazard development up to the present-day situation.

1 Introduction

This paper gives an introduction to the latest developments in modern electronic design methodology. It will give a brief history of the evolution of design software in an attempt to explain the seemingly haphazard development of these tools to become what is now known as the Electronic Design Automation industry. The emphasis is to provide a common background to all school attendees with specific references made to the tools used during the week.

The frenetic pace of advance in the electronics industry is not expected to change in the near future. According to the research firm Gartner, there is no slowdown in the expected revenue growth for the Field Programmable Gate Array, Application-Specific Integrated Circuit and Application-Specific Standard Product markets for the next few years [1]. FPGAs look set to maintain the highest compound annual growth rate (almost 20%) as more and more applications adopt this solution due to their increasing functionality and decreasing costs and the often prohibitive development costs for ASICs and ASSPs. (ASSPs are essentially ASICs dedicated to a specific application market and sold to more than one user.) The total 2007 revenue from three families is expected to reach almost 100 billion US dollars.

Modern-day electronic design tools have evolved for a very good reason. The increasing size, speed, and complexity of these new products and the increasing pressure on bringing them to market quickly means that design targets would be impossible to reach without help from software programs.

Tool development has been continuous over the last thirty years. What traditionally was based on the two main axes of analog or digital designs has now been expanded to include software additions and we shall see how these previously distinct areas are becoming increasingly blurred and merging.

1.1 A brief history

Before electronic design automation, PCBs were designed by hand and manually laid out. Perhaps more surprisingly, this was also true for IC design. The first programs developed were drafting software that included some digitizing capabilities with different companies focusing on specific areas of the market. Calma was a major player in the area of IC manufacture (its GDSII format still exists) while other essentially similar products targeted the PCB sector. While invaluable in laying out a board or IC, there was still no link between the design process and the eventual layout. Various large companies developed their own in-house software and methodology solutions and some of these teams were eventually spun out as separate companies in the beginning of the 1980s. Various pioneers of the EDA industry date from this period: Mentor Graphics (original parent company Tektronix), Daisy Systems (Intel) and the forerunner of Cadence Design Systems, Valid Logic Systems (spawned from Hewlett Packard and the Lawrence Livermore Laboratory).

The need to simulate the transistors being laid out for all ICs was one of the inspirations for the development of the Simulation Program with Integrated Circuit Emphasis (SPICE) software at the University of California, Berkeley. The main driving forces behind the original SPICE program are credited to Larry Nagel working under the supervision of Donald Pederson. The work was based on the CANCER program and was one of the many such projects funded by the United States Department of Defense. Pederson insisted that the newly written SPICE be far enough removed from the original CANCER such that any distribution restrictions could be removed and that the program be put in the public domain.

The first public presentation of SPICE1 at a conference was in 1973 [2]. This was a relatively limited-feature program based on nodal analysis (leading to problems in describing some circuit elements, particularly inductors) and fixed time-step transient analysis (this aspect will be discussed in more detail later in this paper). SPICE2 was introduced in 1975 and offered improvements such as more inherent circuit elements, variable time-step analysis, and solutions based on modified nodal analysis. These were significant advances and ultimately led to the program's widespread adoption. SPICE2 was further developed until 1983 with the introduction of SPICE 2g6 which was the last FORTRAN version of the program. SPICE3 was written in C and was introduced in 1989.

The insistence on SPICE's open-source development (and famously, program availability for the cost of a magnetic tape) led to its becoming widely distributed and used as the basis for countless other simulators in the academic, industrial, and commercial worlds. Well-known commercial versions still existing today include HSPICE (now owned by Synopsys) and PSpice (currently owned by Cadence Design Systems).

Most 'digital' design tools also being developed during the same period were largely based on schematic-driven design entry. One of the next major advances was the development of text-based Hardware Description Languages (HDLs) describing design behaviour at the Register Transfer Level. RTL describes the operation of synchronous digital circuits where signals are treated before being clocked into hardware registers. HDLs allowing circuits to be described using RTL provide a much higher level of abstraction than at transistor or at gate level. Since the strengths and advantages of this level of abstraction were realized, tools were subsequently developed to allow direct and automatic gate synthesis from the HDL circuit description. This was a major step forward in the EDA industry and allowed a huge advance in design complexity and reduced time to market.

Two major HDLs became prominent during the mid-80s.

Verilog is a proprietary language defined by Gateway Design Automation. This was an industry changer and is still the prominent language in certain design areas. Written by Phil Moorby in around a year, it quickly became the de facto standard for ASIC design along with the Verilog-XL simulator. Its impact was reinforced when Synopsys introduced software permitting direct gate-level synthesis from the Verilog netlist. This allowed a direct line from design description to simulation and layout in one integrated flow. Gateway was eventually acquired by Cadence Design Systems during 1989. With the increasing market penetration of VHDL (described below), Cadence decided to make the language generally available through the Open Verilog International organization. The language was later submitted to the IEEE for standardization and this was eventually accorded as IEEE Standard 1364-1995. Subsequent developments of Verilog have culminated with the release of SystemVerilog.

The Very-High-Speed Integrated Circuits Hardware Description Language (commonly abbreviated to VHDL) was developed for the United States Department of Defense. It was becoming impossible to reproduce reliably the functionality of technically obsolete equipment as its specifications were insufficiently documented. VHDL was conceived as an attempt to overcome this limitation by defining a language that could formally and completely describe an object's specification. It was based on the existing ADA language syntax so as to benefit from already well-proven concepts. As with Verilog, the idea of being immediately able to simulate and synthesize the specification was conceptually attractive and appropriate logic simulators and gate synthesis tools were subsequently developed.

Unfortunately, the advent of these two similar but competing languages led to a 'Dark Ages' period in the EDA world where it was unclear which language would prevail. VHDL was touted as the new 'standard' language in the industry but the already existing base of Verilog users guaranteed that Verilog remained and continues to be widely used.

The mid-80s also saw the introduction of Field Programmable Gate Array circuits. Originally developed by a Xilinx co-founder, the FPGA was based on complex programmable logic devices but offered more design flexibility at the expense of more difficult design effort and timing uncertainties. This type of circuit could be described by the existing HDLs along with the newly written synthesis programs. As noted in the paper's opening section, FPGAs have become an important part of the market and seem set to at least maintain their share in the next few years.

1.2 Mixed-signal modelling

Modern circuitry often includes both analog and digital circuitry. The most obvious example of this is in a mobile phone where there is a radio-frequency front-end involving amplifiers and mixers before the downshifted signal is treated using digital signal processing. To simulate this complete circuit, some form of mixed-signal simulation is needed. In the extreme case, all circuit elements could be modelled and simulated at transistor level in SPICE but this would soon lead to unacceptably long simulation times for any design of complexity. Also, SPICE simulates a circuit by resolving Kirchhoff's current and voltage laws and as such does not understand any signals that are non-conservative, e.g., signal flow. The program also supposes continuous time, so does not support the idea of discrete-event simulation. These are a few of the problems facing mixed-domain simulation solutions.

Digital-only simulators can easily perform many simulations very quickly. There is a time clock when all events are scheduled to happen, all circuits can be described as a list of time-scheduled queues, and all events are predictable.

A modern SPICE simulator faces a completely different set of problems. Knowing the value of a function at some point in time, it has to estimate the function at a future time. It does this by calculating an approximation to an analytical solution at discrete time points using numeric integration. One method would be to use the forward Euler method: from a given point, the next point is estimated from the slope of this known initial point:

$$y_{n+1} = y_n + hf\left(t_n, y_n\right) . \qquad (1)$$

This explicit method is mathematically very efficient but can, however, lead to stability problems. The backward Euler method is instead often used:

$$y_{n+1} = y_n + hf\left(t_{n+1}, y_{n+1}\right) . \qquad (2)$$

This implicit method is computationally more demanding but leads to more stable results. One method of calculating the value is to expand the Taylor series around the original point. For completeness, we mention that SPICE simulators often use the Runge–Kutta algebraic approximation to the Taylor series to evaluate this new value. This method is commonly used in other commercial programs such as the Simulink simulator that will be used during this school.

To reduce simulation times, SPICE uses a variable time-step during analysis. During periods of fast signal change, more calculations are needed to produce what is deemed as tolerably accurate results. During periods of slow signal change, the time-step can be larger while still generating acceptable solutions. SPICE determines the ‘goodness’ of its results at each time-step from an evaluation of the resolved node voltages and currents:

$$\left|V(n) - V(n-1)\right| < VLimit = V(n) * Reltol + Vntol \ . \tag{3}$$

$$\left|I(n) - I(n-1)\right| < ILimit = I(n) * Reltol + Abstol \ . \tag{4}$$

If these inequalities are not satisfied, SPICE will reduce its time-step and again resolve the equations to produce another approximation (this recursive action is sometimes called ‘rollback’). If the simulator’s time-step can eventually not be further reduced to produce an acceptable solution, the solver will fail and signal that it has failed to converge. Experienced SPICE users will recognise not only the message but also the parameters used in Eqs. (3) and (4). The values for *Reltol*, *Vntol* and *Abstol* can be adjusted (with care) from within the simulator to overcome such convergence problems.

The preceding paragraph highlights the problem of performing mixed-signal simulations — digital simulation is done using fixed, well-defined time-steps while analog simulation uses variable time-steps and getting both simulators working together is not trivial for efficient mixed-signal simulations. The situation is complicated by rollback where it is possible that a re-calculation of the SPICE values can affect what were previously correct digital values. Some techniques have been used to overcome these problems:

- Lockstepping the simulators such that the two engines are locked together in time throughout the simulation. This means that the simulation time is dependent on the smallest time-step and can lead to very inefficient use if the analog circuit is non-trivial.
- Using ‘backplanes’ between simulators where each simulator connects to a virtual software interfacing bus.
- Other proprietary techniques, e.g., the Calaveras algorithm. This allows one simulator to run ahead of the other. If the simulators subsequently determine that evaluations in one domain would have affected the other, the affected simulator ‘rewinds’ and repeats simulation with the new data. The technique can often lead to the discarding of large amounts of data which again reduces efficiency.

Despite these difficulties, mixed-domain simulation solutions are available and indeed, both the Verilog and VHDL languages have been extended to include ‘Analog and Mixed-Signal’ capabilities (Verilog-AMS, VHDL-AMS).

1.3 System-level design

The necessity for mixed-signal simulation led to the development and use of such solutions. However, modern designs can also involve a large amount of circuitry along with a large amount of embedded software (the mobile phone is again an example). Traditionally, hardware and software have been developed concurrently but distinctly from a single specification. This methodology has several drawbacks at different levels. Some of them follow:

- The translation of the specification requirements to the hardware and software branches is done separately and manually.
- What is being tested in the hardware and software implementations is what has been understood as the design intent after this translation — this might not be the same as the original specification.

- There is no obvious link between the hardware and software.

These limitations have been appreciated for many years and have inspired the development of *Electronic System Level* tools. Unfortunately, ESL design does not have a well-defined meaning but most people seem to agree that the concept makes possible:

- fast exploration of the solution space before detailed simulation (top-down design)
- easy change of the level of model sophistication
- co-simulation of hardware and software system behaviour

One other great advantage of a system level design would be if the original specification were executable. This is conceptually a very attractive idea — the specification itself would act as the model that can be used in the design and by other tools for simulation, verification and eventual synthesis. This obviously removes one important drawback in the traditional flow where what is built is what was perceived as the intention from the specification and not the specification itself.

The flexibility and availability of a language such as C++ would seem to make it a good basis as a system level design language. However, C++ is based on sequential programming so it is unsuited to modelling concurrent processes and also lacks the notion of time — both notions being essential to model successfully at the hardware level. Other essential features for system modelling such as hardware data types and definitions for signals and ports are also missing.

Attempts to remove these limitations have been made in SystemC. SystemC defines a set of libraries for C++ containing the constructs and core language elements needed to perform hardware modelling in C++. This allows hardware to be described using C++ syntax and to be compiled into an executable that will behave as the modelled hardware when run. The required SystemC libraries have been made generally available by the Open SystemC Initiative. OSCI also makes available a reference simulator, although arguably more advanced commercial versions are available. EDA vendors also offer mixed-language possibilities allowing SystemC models to be simulated with previously defined Verilog or VHDL models. This improves performance as SystemC simulation speed at RTL level is not yet at the level of other HDL simulators.

Different levels of model abstraction can be described in SystemC (though is not limited to SystemC) through Transaction Level Modelling. TLM enables modelling at a far higher level of abstraction than RTL. Using TLM, the communication between modules is separated, and can be developed independently from the description of the functionality of the modules. Briefly, channels are used to model communication between components and transaction requests occur by calling interface functions to these channels. Taking the example of a design's system bus, this would be represented by such a channel defined as part of a SystemC interface class. Transaction requests then take place through these channel models through this interface. This makes it easy for the designer to explore different bus architecture possibilities as long as all models interact with the different bus representations through the common interface. The use of this common interface also allows models to be easily refined to include more functional and timing detail. The different levels of refinement are often called views and which one was used during simulation depends on the level of detail required at that design stage — the more detailed the model, the longer the simulation time. Unfortunately, there is no standard TLM nomenclature but one set of definitions is

Functional View – The design focuses on algorithm development and not implementation. Timing is not a concern.

Programmer's View – The hardware models are functionally correct but timed only with sufficient detail to allow use by the software developers.

Architect's View – The level at which architectural exploration and optimization is conducted. Exploration is again done using approximate timing.

Verification View – The hardware models are refined to include timing detail and are cycle-accurate.

Figure 1 graphically represents the different available possibilities for TLM refinements (at the time of writing of this paper, the TLM-2 Draft was under discussion by OSCI to try and define at which abstract-level each view should be modelled).

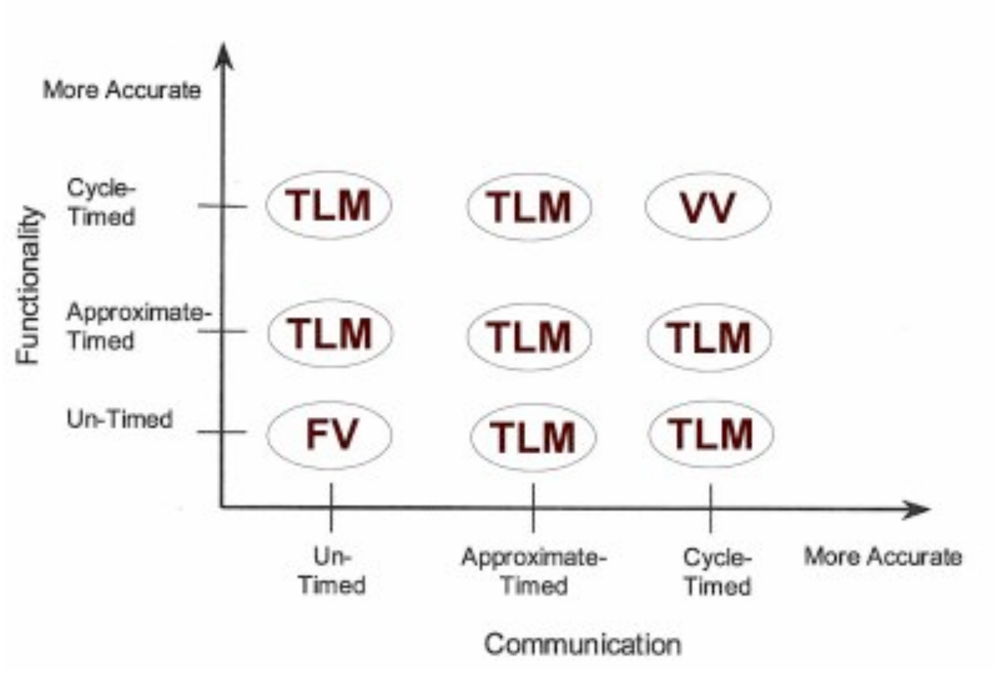


Fig. 1: Abstraction level for transaction level modelling

MATLAB is another program that is widely used and can be considered as an ESL language. Typically used for DSP and control algorithm developments, it has suffered as there was no easy method of translating proprietary MATLAB or Simulink code into equivalent RTL. However, recently introduced toolboxes and applications have eased this problem, making for a more complete design flow. Third-party applications (such as Xilinx's SysGen as used in this school) have also been developed that allow direct implementation of The Mathworks Simulink designs into FPGAs.

Other existing languages used with different levels of penetration include Rosetta, Esterel, Specification and Description Language, XML and UML.

1.4 Design verification

The ultimate aim of verification in the case of electronic design is to check if a completed design conforms to the original specification. The overall verification procedure should answer the following questions:

- What is the design intent?
- What does the design actually do?
- Do the two things match?
- How good is the testing?

With increased design complexity, a substantial part of the overall design effort is involved with circuit verification. Some estimates place it as high as 70% and large companies often have separate verification teams that are involved only with this aspect of the project. These verification teams often work separately from their design colleagues as they are essentially involved at different levels of testing. The verification engineers work at 'black-box' level. They are interested only with verifying behaviour against the original specification and are unconcerned with the actual design implementation. Design engineers work at the 'white-box' level; they understand fully the inner workings of the design and test at a more detailed level. While seemingly an inefficient way to operate, this working method can help to highlight any differences in the interpretation of the original specification.

Simulation has traditionally been the most common method used to test a design but this method suffers from some disadvantages:

- Setting up a testbench to verify each desired functionality can be time-consuming and a non-trivial matter.
- Writing a full testbench to cover all possible states for large designs is practically impossible.
- No indication of how well the design has been tested.

To help overcome these problems, special verification languages such as Vera and Property Specification Language have been developed to allow easier and fuller design testing, often using constructs called *properties* and *assertions*.

A *property* is a Boolean expression defined in the syntax of the HDL used to describe the model and can also include some extra temporal information.

An *assertion* is used to ask a verification language to evaluate a given property.

The following is a simple PSL property example.

```
property Nooverflow is always (Mydata < 256);
```

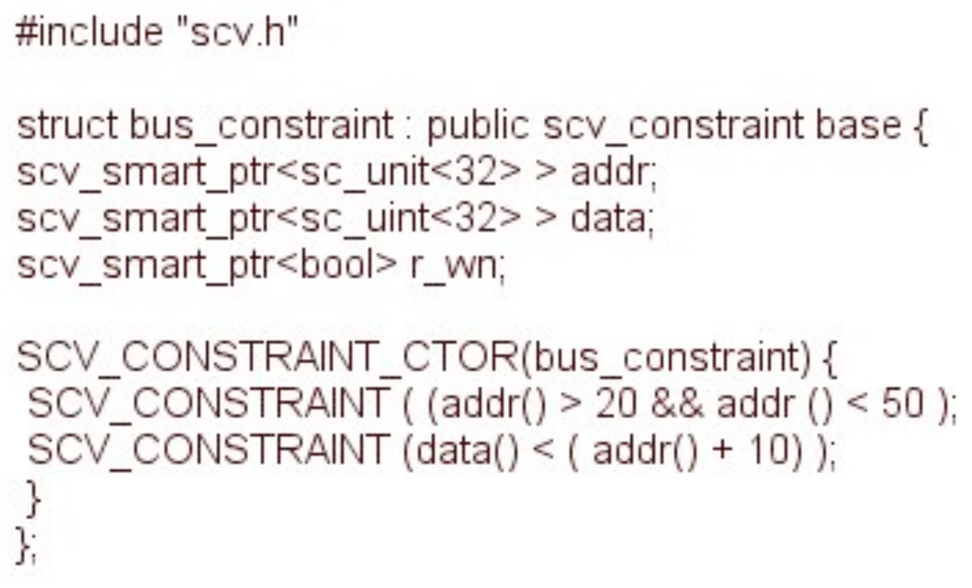
We have defined a property *Nooverflow* that states that *Mydata* must be always less than 256 (*always* is a temporal keyword in this example). We can then include an assertion in our code to ask that the simulator verify that this is true:

```
assert Nooverflow;
```

Properties and assertions can be used during simulation to check for specific behaviour and are a very valuable addition to the available tools and techniques. However, verifying assertions adds simulation overhead so care should be taken to use them at an appropriate level.

Generally, the more different the test performed during simulation, the better the verification coverage. Verification tools can be set up to create automatically a set of testbenches to improve the coverage while relieving the designer of the task. A random seed is used as an input to the automation process and a testbench created from this arbitrary value. Properties and assertions can then be used in this context to check if this newly created testbench is valid or not.

Figure 2 shows an example using SystemC (and its associated verification capabilities) where the addr variable is limited between 20 and 50, and data has to be addr+10.



```
#include "scv.h"

struct bus_constraint : public scv_constraint base {
scv_smart_ptr<sc_unit<32> > addr;
scv_smart_ptr<sc_uint<32> > data;
scv_smart_ptr<bool> r_wn;

SCV_CONSTRAINT_CTOR(bus_constraint) {
 SCV_CONSTRAINT ( (addr() > 20 && addr () < 50 );
 SCV_CONSTRAINT (data() < ( addr() + 10) );
 }
};
```

Fig. 2: Example of constrained values using SystemC verification libraries

Constrained random data generation can help enormously in setting up and running a large number of simulation tests. However, it is impossible to know which addresses and data values are generated until the test is run. Recording the actual values created helps to provide verification coverage metrics. Assertions can also be included to determine if a particular test verified a given design feature, and to what extent. This information can again be recorded to help provide a more complete coverage metric. The gathered data can then be used to determine if all cases have been covered and whether any more tests should be run to complete the testing. This overall procedure is known as coverage-driven verification.

While CDV can help to cover more possible simulation cases, it is still not an exhaustive test. A technique called Formal Model Checking can be employed to guarantee full coverage.

FMC performs a mathematical analysis of all the possible states of a design to determine if any violate the included properties or assertions. This technique is a full, rigorous, formal analysis so there is no need to write a testbench. While it would be theoretically possible to use FMC to verify fully the functionality of any arbitrarily complex design, it is typically used for sub-blocks of a design and used in conjunction with simulation. If FMC has been used to prove that a given sub-block behaves correctly for a set of inputs and the sub-block is only driven by inputs in this range during simulation, the sub-block can be assumed to function correctly in the complete system. For completeness, it should be noted that in some cases, the same assertion used in a simulation run and in a formal analysis verification can return different results, e.g., it will pass in one test and signal a failure in the other analysis. Tool vendors overcome this problem either by supplying carefully defined assertions that are known to be consistent in both cases or by cross-checking the assertions by running both analyses.

An interesting variation using both FMC and simulation is known as Dynamic Formal Verification (DFV). This is particularly useful for known ‘dangerous’ cases, e.g., FIFO is full. For example, using constrained driven verification only, FIFO_full might be tested but some other condition at the same time could cause a bug (it would be unlikely that both conditions are present using CDV alone). DFV recognises that this hazardous situation has occurred and then instigates a formal analysis on all states leading up to and beyond this time to identify any possible problems. This helps to target verification time on potentially dangerous situations.

Recording the outputs of the analyses above is used as an input to provide a functional coverage metric. This can be used to determine when we have performed 'enough' tests. Another metric used to assess verification completeness is Code Coverage. This is invariably measured as it is conceptually very simple and essentially reports how much of the HDL code has been exercised during testing. This is a rather simple test but can guarantee that no dead code is synthesized. While easy to gain a set of metrics for the coverage, it can unfortunately return an artificially high pass-rate, for example, exercising a circuit's 'reset' can cause many lines of code to be accessed. These lines will therefore be deemed as having passed though there is no guarantee that they will be exercised again during the verification process.

Code and Functional coverage tools should be considered as complementary in the verification procedure.

2 Conclusion and future developments

We have seen how tools have evolved to solve the immediate problems faced by design engineers. We have also seen how the gradual merging between analog and digital hardware and software designing is gradually taking place. Industry has also recognized the difficulties faced to verify sufficiently the very large designs currently being manufactured.

Specialized design point-tools such as MATLAB have become important in some niche areas. They have traditionally been used as algorithm development tools only, but their capabilities have recently been extended to allow direct FPGA implementation from the model description.

SystemC has been mooted as an industry-standard solution covering all of the above design aspects. It provides a good system level design solution and, since its introduction, has been extended to include verification capabilities using SCV. It has a strength in that it is founded on C++, a well-known and widely-used language. However, simulation at hardware level is much faster using one of the standard HDL languages and their associated simulators. Hardware engineers also are not necessarily familiar with C++ and this might contribute to the fact that SystemC uptake is currently not as extensive as was once forecast.

The verification bottleneck has led to the development of specialized languages focusing on this problem. These have become rather well used (due to necessity) but they again tend to suffer from the fact that their syntax is different from the underlying HDLs.

Verilog and VHDL are continually being revised to address the new problems faced by the industry.

VHDL is currently being considered to include stronger verification capabilities.

Verilog has been extended to SystemVerilog which became an IEEE standard in 2005. SystemVerilog combines not only Verilog language updates (including stronger typing as found in VHDL) but also includes system-level and verification capabilities. It is significant in that it is the industry's first unified hardware design language — for the first time a common HDL can be used to cover all aspects of the design flow from description to verification.

This could lead the language to become the de facto industry-standard HDL but, as ever, this remains to be seen.

Acknowledgement

Extensive use has been made of the Internet and various texts to prepare this paper but this article is presented as original work. Any errors or omissions of acknowledgements are entirely accidental and, should any have occurred, full apologies are extended. Various well-known commercial tools have been mentioned in this article and their trademarks and any other copyright rights are acknowledged.

The author would like to thank Synplify, The Mathworks, and Xilinx companies for graciously offering their software for use during this school and for their help in installing the programs.

References

[1] http://electronicdesign.com/Articles/Print.cfm?AD=1&ArticleID=14442.

[2] L.W. Nagel and D.O. Pederson, SPICE (Simulation Program with Integrated Circuit Emphasis), Memorandum No. ERL-M382, University of California, Berkeley, Apr. 1973.