Time to exploit IDEs for hardware design and verification

Cristian Amitroaie, Amiq Consulting, Andrew Betts, an independent consultant

5/18/2011 10:26 AM EDT

Integrated Development Environments are solidly established in the software community. Eclipse, a major open source IDE, has been downloaded over 5 million times. Data published in 2008 by its rival, NetBeans, showed their IDE to be in use by over 2 million users worldwide. Commercial IDEs, such as Microsoft's Visual Studio, are also popular.

The IDE's advantages to software engineers are many, and most of them are relevant to engineers working on hardware verification. It is, after all, a software task. Languages used include C, C++ and SystemC, of course, but increasingly the dedicated verification languages *e* and SystemVerilog. Many aspects of hardware design are also language based, and can potentially benefit from the use of an IDE. Most digital designers use VHDL or Verilog, and Analog/Mixed-Signal (AMS) designers are increasingly taking up related languages such as VHDL-A and VerilogAMS.

By taking a closer look at the differences between the general software and the more specialized hardware verification communities, this article will attempt to explain the current growth and future prospects of IDEs in hardware verification. The implications for hardware design are also examined.

The modern IDE
A modern IDE is much more than a glorified editor, both in terms of features and architecture. Furthermore, highly optimized graphical interfaces make initial use quite intuitive, allowing advanced functions to be learnt as the need arises (more on this below).

Figure 1 presents a conceptual snapshot of an IDE's main functions, with emphasis on the flow of information to the user from multiple sources. In addition, an IDE acts as a central point of control over tools in the environment. A key component of the IDE is therefore the context sensitive user interface. This must display the information and controls that the user needs at any point in time, while suppressing distracting information.

As well as being convenient, the IDE's ability to control the toolchain, including simulators, has the advantage that different tools may be plugged into the system without changing the way in which they are driven. It therefore decreases the burden of learning new tool interfaces and increases the tool/vendor neutrality of the flow.

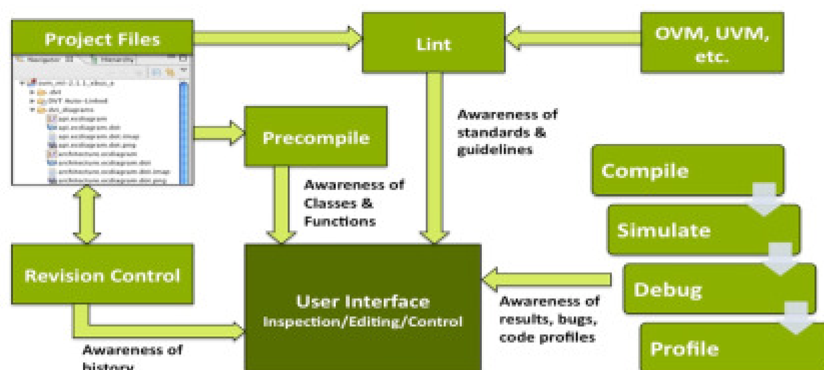

Fig.1: IDEs provide all-round project awareness and control

When oriented towards code development, a platform such as Eclipse may include, for example, toolchain management (compile, run/simulate, debug, revision control), code profiling and linting, function and class browsing/navigation, refactoring support (obsolescing terabytes of grep-sed-awk scripts!) and, of course, syntax and semantic checks with auto-completion. Please see the sidebar for more information on typical IDE features.

The fundamental, differentiating benefit of such systems, when compared to file editors and other file processing tools, is that they are aware of and manage the relationships between many different aspects of the design. That is, they operate on a project basis, relating multiple information sources to the file being worked on and communicating this to the user. Once an engineer has experienced the advantages of such a system over simple file-by-file editing, there is no looking back!

Traditional barriers to IDE adoption in hardware verification

Given these impressive features and benefits, why have IDEs not been used more widely in the hardware domain and, in particular, in hardware verification, which is essentially a software task? We believe that there are three main reasons:

- The level of complexity in a typical hardware verification environment is significantly less than in many pure software projects (operating systems, financial control, games, etc). Not only is the code base very large in the latter projects, but so are the associated libraries of reusable software IP.
- Early open-source plugins for hardware design and verification languages (VHDL, Verilog, *e*, SystemVerilog) were disappointing and poorly supported.
- Hardware verification engineers have been reluctant to invest time in learning how to use IDEs. A verification environment is complicated enough already, with multiple compilers, simulators and languages, and it is a constant struggle to minimize the number of tools used.

Taking these points in reverse order, and addressing the issue of the IDE learning curve first, we note that IDEs have come a long way in the past few decades. Years of experience with many and varied products help the designers of new IDEs to produce highly optimized use models. Also, fierce competition between alternative IDEs drives this optimization.

The tools that have emerged from this process are therefore extremely user-friendly, and someone using an IDE for the first time can begin with just the basic functions of the system. A modern, enlightened IDE will not be insulted if you use it as a simple editor! Even with this approach, one benefits from auto-completion, syntax highlighting etc, since the tool will recognize the language being used automatically. In this way, a verification engineer can begin to reduce the number of tools that he or she has to manage directly – the IDE will first replace the user's old editor, then later displace, for example, sed, grep, awk, and make. It can also front-end source code control tools such as SVN and CVS, as well as managing calls to compilers, simulators, debuggers etc. Hence, although the latter tools do not disappear from the flow, their use becomes transparent once they are integrated into the IDE.

It sounds wonderful but, as mentioned, early, open source efforts to support verification languages in IDEs were disappointing. The reason for this, we believe, lies in the relative size of the software and hardware communities. Whereas Eclipse and NetBeans alone account for many millions of users, we estimate that the number of hardware design and verification users to be a few thousands only. In fact, the total number of hardware design and verification users is probably less than the number of open source IDE software developers in the world – Eclipse alone boasts around 200 open source projects, close to 1,000 "committers" (people that submit software updates), and over 160 member companies (see **www.eclipse.org**).

The hardware community is simply not large enough to support open source style IDE projects - commercial solutions are a much better bet in this case. Fortunately, these now exist for all the main languages (VHDL, Verilog, *e*, SystemVerilog).

Open source is a good thing for the hardware community however. Many commercial language plugins, including the most important ones in the hardware domain, are based upon them. Such products immediately benefit from a huge database of pre-existing code, so that the resulting IDE systems are feature-rich and relatively bug-free. The availability of fully supported, commercial products has therefore removed a major barrier to the adoption of IDEs by the hardware community (see, for example, **http://www.dvteclipse.com**).

The final obstacle to adoption, listed above, was the relatively low complexity in hardware verification code, contrasting with the situation in software projects, which quickly run into hundreds of thousands of lines of code, plus massive associated libraries. Hence, the argument goes, verification engineers do not need help from an IDE to develop and manage their code.

But this argument is now outdated.

The pressing need for IDEs in hardware verification

Modern verification languages, such as e and SystemVerilog, are themselves significantly more extensive than the VHDL and Verilog languages used to design the circuits themselves. Further, a hardware designer mainly uses a synthesizable subset of either VHDL or Verilog, and so the language possibilities are even more restricted. *e* and SystemVerilog, on the other hand, are complex behavioral languages with special constructs to support hardware testing. These include constrained random test generation, coverage, events, temporal expressions and assertions. Using such constructs considerably simplifies the task of creating, for example, bus functional models, scoreboards and signal monitors.

In fact, it is not uncommon for the code of a verification testbench to outweigh the code of the Design Under Test (DUT) – in terms of lines of code, compilation time, runtime, or any other common measure. This stands to reason when one considers that the verification testbench usually contains a complete functional model of the DUT (as a reference), together with constrained-random test generation modules, monitoring systems, coverage measurement (sometimes with dynamic feedback to the test generator) and reporting. In addition, there is often a lot of redundant code in a verification system, since testbenches must be as reusable as possible, and therefore applicable to multiple, similar DUTs. All in all, there is a lot of code to manage.

And if the size of the languages and the testbenches were not enough, there is now a growing body of verification IP and associated standards that anyone coding a verification testbench must take into account. Unlike silicon IP, verification IP does not have hard or firm variants – it is always soft, and there is therefore a significant code database to assimilate and reuse. An IDE can be a tremendous help when dealing with incoming verification IP. It can quickly provide statistics on the IP package, allow navigation of the code and its class hierarchy, and uncover any inconsistencies with the user's coding standards.

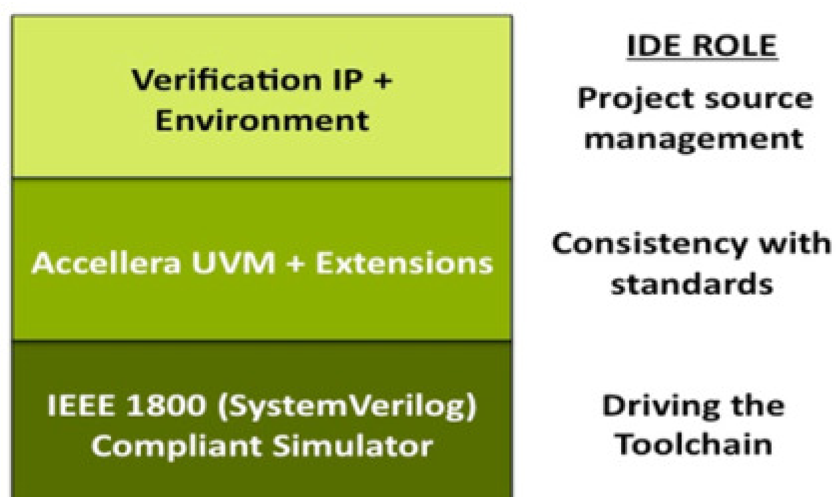

Fig.2: UVM layer diagram showing the role of an IDE

Nowadays, the user's coding standards are likely to be built on those of the Universal Verification Methodology (UVM), or one of its predecessors (e.g. VMM and OVM – ***see sidebar***). The class libraries and methodologies associated with these standards have become important references for hardware verification with e and SystemVerilog (***see the sidebar "Hardware Verification Standards and IDEs"***). They are a promising step in the direction of increased verification IP reuse and, in particular, simulator neutrality for testbenches (to allow users to port their testbenches more easily between simulators from different vendors). However, they also represent more "stuff" for verification engineers to cope with. An IDE can greatly assist users in using these libraries and following the methodologies, and IDE plugins that check against these standards are commercially available.

Three adoption scenarios
It can therefore be seen that the code complexity factors that have made IDEs popular in the software community are now strongly present in the world of hardware verification. Since the former obstacles to their adoption in the hardware domain have been substantially reduced, we expect them to become much more widely used, first by verification engineers and then by designers.

Let us finish by considering three scenarios where the use of a modern IDE can add significant value.

Overworked verification engineers are the most obvious IDE beneficiaries. For them, mastering a powerful IDE can be a key career investment. Of course, this argument also applies to design engineers, especially those that build complex testbenches. Nowadays, both design and verification engineers must routinely master multiple computer languages, libraries and standards, and a good IDE helps keep all this under control.

A second scenario is the technical manager, whose contribution to testbench coding may be limited or even zero, but who needs to be able to share an understanding of design and verification issues with his team, measure progress and anticipate crises, and also assess external verification IP. An IDE, which is inherently a project-oriented (rather than file-oriented) tool is superb in this role.

The final scenario is the consultant, internal or external, who is called in to help on a verification project, and therefore has to take stock of a large body of code in a short time. The IDE's ability to profile and navigate the code is essential to him. Also, for example, its refactoring capability, which allows someone to make large scale changes to a code base with much reduced risk.

Our overall conclusion is that IDEs will inevitably see wider and wider use in the hardware verification community and also, with a small delay, in hardware design. Although the adoption of IDE technology in these domains is significantly behind that in the general software community, the rapid development of hardware verification languages, IPs and standards is a strong driver for their use. Thanks to a combination of progress in open source IDE platforms and the availability of commercial plugins for key languages and standards (*e*, SystemVerilog, UVM, OVM, VMM, VHDL, Verilog), early obstacles to adoption have been substantially reduced. Commercial products based on open source foundations are most likely to provide the best cost/benefit compromise, and this is where we expect to see strong growth over the next few years.

SIDEBAR 1: Typical IDE features
Project scope. The IDE knows about the entire software project, and can therefore give advice and run checks related to all the files and libraries involved. This includes knowledge of class hierarchies, module instances and connectivity.

Static code analysis. The project code is pre-compiled in order to obtain information about the content of the project and dependencies. As well as allowing many different types of checks (*see below*), this can also free up simulation software licenses, thus reducing project costs.

Real-time syntax checking and auto-completion. Using its knowledge of intra-project dependencies, the IDE can prompt the user with the correct syntax and possible parameter, function,

object and class references.

Refactoring helps the user to change the name of an object in the software system. The IDE uses both syntactic and semantic knowledge in order to change, for example, the name of a function call, propagating the changes to all the affected files in the project.

Profilers built into IDEs allow the user to obtain both a static and dynamic overview of the project. A static overview includes, for example, the number of lines of code, the number of functions and the depth of class hierarchy. A dynamic overview shows how runtime is distributed through the lines of code in the project when the software is run.

Built-in debuggers allow users to insert breakpoints, trace the program flow, display changes in parameter values, etc.

Class browsers provide a kind of auto-documentation feature, typically displaying class hierarchies with short documentation headers derived from the code.

Linter systems check not only for proper style, but also with reference to standards (for example, company or industry standards, such as UVM, OVM or VMM in the case of hardware verification).

Toolchain links allow users to run code directly from the IDE, as well as manage code through associated source control systems.

SIDEBAR 2: Hardware verification standards and IDEs

With the right plugins, IDEs can be used to check not only language syntax and semantics, but also methodologies. Checking is based on the rules and class libraries associated with these standards. Here are some key references that can be supported by IDEs working in the hardware verification domain:

e: The *e* hardware verification language is supported by Cadence's Specman™ product (originated by Verisity). Introduced in the mid 1990's, it was the first widely adopted language to support constrained random test generation and coverage measurement. The *e* language is described by the IEEE draft standard, P1647.

SystemVerilog: SystemVerilog is a combined hardware design and verification language. A superset of Verilog, it was born out of the SuperLog language, donated to Accellera in 2002, and it adopted much of its verification functionality from OpenVera. The influence of Verisity's e language, initially a strong competitor of Synopsys's Vera, should not be overlooked. Verisity was acquired by Cadence in 2005, and SystemVerilog became the IEEE 1800 standard in the same year.

VMM: The Verification Methodology Manual was published by Synopsys and ARM in 2005. It defines a coverage driven, constrained-random methodology for implementation using SystemVerilog, for which it defines a base library.

OVM: The Open Verification Methodology was the result of joint development between Cadence and Mentor Graphics to develop a standard library and a proven methodology based on SystemVerilog. OVM's ambitions included SystemVerilog interoperability between simulators and an acceleration of the development of verification IP written in SystemVerilog, SystemC® (IEEE 1666), and e languages. The first OVM release appeared in 2008.

UVM: The Universal Verification Methodology standard was developed by Accellera and the 1.0 version was released in February 2011. It is available as a Class Reference Manual accompanied by an open-source SystemVerilog base class library implementation and a User Guide. UVM is a direct derivative of OVM and has essentially the same objectives. It has also drawn material from VMM.

About the authors:

Cristian Amitroaie is co-founder of Amiq Consulting (**www.amiq.ro**) and, more recently, Amiq EDA (**www.dvteclipse.com**).

Andrew Betts is an independent consultant working in the area of Technical Sales and Marketing for EDA (**www.icondasolutions.com**).