Verification requirements for SystemC/C++ designs

By Vlada Kalinic, SystemC Verification Product Manager, OneSpin Solutions

Although SystemC/C++ coding styles have been used for many years, specific models have recently emerged to drive common design flows across engineering teams. These include abstract algorithmic design code as input for high-level synthesis (HLS) tools, virtual platform models for early software test, configurable intellectual property (IP) blocks, and many more.

HLS, which transforms "mostly untimed" abstract SystemC/C++ design representations to fully-timed register-transfer-level (RTL) design blocks, is in use at many large semiconductor and electronic systems companies. These tools are particularly popular as a method to rapidly generate design components with varying microarchitectures, whilst rapidly and effectively optimising algorithm-processing data paths. Their use on control logic, as well as components with more detailed timing in general, is also becoming more widespread.

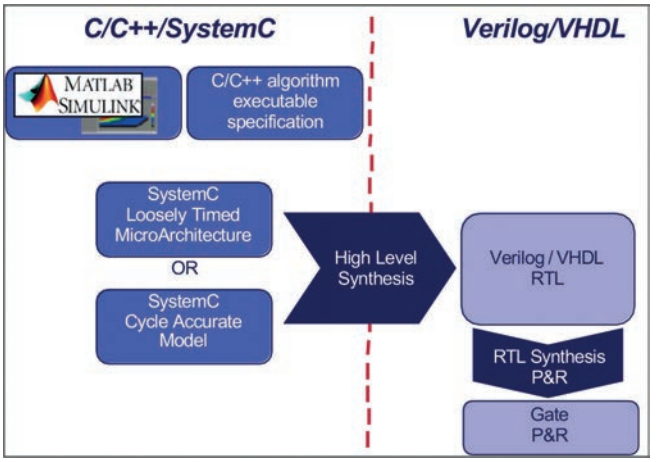

Figure 1: SystemC/C++ high-level design flow

Verification of System C/C++

The verification of SystemC/C++ designs is largely performed by compiling the design representation using a standard software compiler, such as GCC, and debugging the code in a similar fashion to software designs. The Open SystemC International (OSCI) SystemC/C++ class library, now standardised as IEEE 1666-2011, introduced a functionality that provides a more RTL-simulation-like user experience. Still, there are many issues that make the verification of SystemC code a complex, arduous task – including debug runtime performance and test complexity. The availability of formal techniques at this level has been sparse.

A common SystemC/C++ HLS flow makes use of algorithmic descriptions often using only C or C++ code. These descriptions are tested to ensure that the algorithm itself operates correctly. The SystemC class library functions are applied to provide minimal hardware detail, such as basic timing, reset functionality, etc., as required by the HLS tools. The synthesis tool produces RTL code, which is then applied to a more traditional design-refinement flow and verification process.

The verification of the design is split between the SystemC and RTL levels. It is clear that engineers would prefer to verify and debug the original SystemC designs, and only check for functional equivalence post-synthesis, similar to traditional RTL development processes. However, the lack of effective SystemC/C++ design verification environments has driven engineers to more traditional HDL verification. As they raise the abstraction level of their design approaches, it becomes more natural to also raise the level of verification. At the pre-HLS algorithmic level, verifying the design directly against its specification with less concern for coding detail

Many designers have moved to SystemC/C++ to raise the abstraction level and take advantage of high-level synthesis

is a requirement. Functional specifications may be represented easily with assertions and, as such, the use of formal techniques is a natural choice since they allow assertions to be rigorously tested against the design. New control-intensive algorithms, now being coded in SystemC, are particularly hard to verify using just simulation.

Going formal

There is a broad range of formal techniques that can be applied to design components coded in C++ or SystemC, with varying levels of timing and code abstraction. Such solutions provide a range of automated structural, safety and activation checks to be applied to designs without having to manually create assertions – particularly useful for sign-off of the design code prior to high-level synthesis.

Fully-functional assertion-based formal verification tools allow for comprehensive assertions to be tested against SystemC/C++ design code. These assertions may be written using the simple C assert statements, or full SystemVerilog Assertions (SVA) with all the temporal concurrent constructs included. The ability to leverage temporal assertions on SystemC/C++ designs is a unique capability of this technology.

Multiple proof engines can leverage a range of standard and proprietary algorithms to provide in-depth code analysis, which consistently exhibits a high degree of convergence compared to other solutions, coupled with rapid, high-capacity operation. The platform can process a range of languages and includes capabilities for easy set-up and usage. A powerful debug environment provides a clear path to quickly track down design problems or tests.

Formal techniques' flexibility allows them to be applied to a range of problems, and these apply equally well to SystemC/C++ designs. Tools may be used in a highly-interactive mode to quickly look at how a design is operating in a "what-if" style. They may form the cornerstone of a full metric-driven verification solution, and an effective validation mechanism for IP integration on SoC platforms.

Automated formal SystemC/C++ design evaluation

The full automated functionality of formal verification may also be leveraged on SystemC/C++ hardware design code. Eliminating bugs as early as possible in the design process can save many engineering hours downstream, and this is even more valuable when the design process starts at the microarchitecture abstraction level.

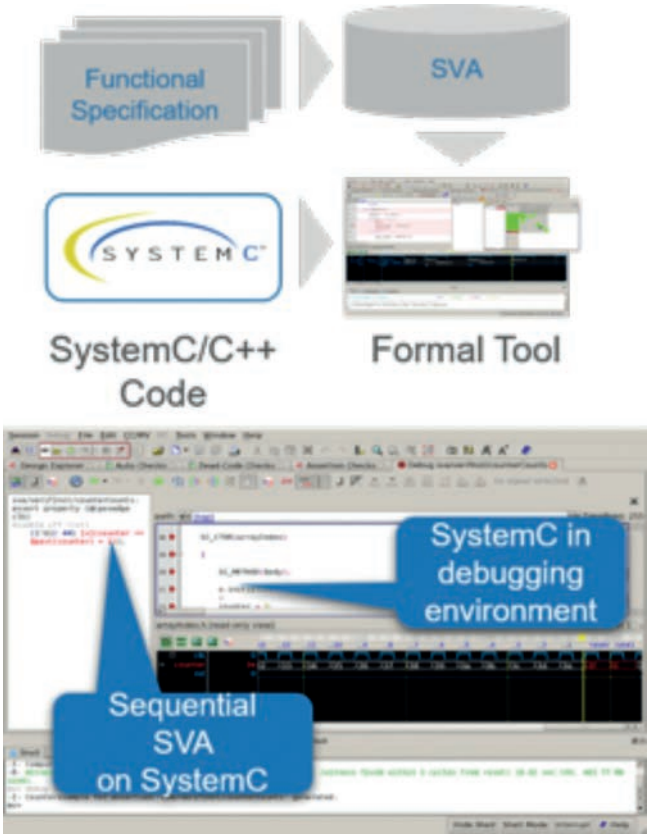

Figure 2: Use of SVA with a SystemC/C++ design

A range of automated checks that use the full power of the formal engines can be implemented to provide in-depth static analysis of the design code, but without having to manually write assertions. Going beyond traditional linting tools (automated checking of your source code for programmatic and stylistic errors), this design inspection technology looks for potential bugs by analysing operational scenarios based on the code construction.

Safety checks such as out-of-bounds access on an array or dead code (unreachable) or deadlock activation checks and structural analysis, including classic mismatches between simulation and synthesis operation, are all available. In addition, there are checks particularly applicable to SystemC code. For example, it is important to check which registers have been explicitly initialised: SystemC variables are automatically initialised in simulation, but HLS tools ignore these initialisations, leading to simulation-synthesis mismatches that are hard to debug. Formal techniques also check to see whether uninitialised registers, undefined operations, or multiple drivers can propagate X (unknown) values in the design.

There is no notion of unknown values in SystemC simulation, so formal analysis is needed to find propagation issues. SystemC also lacks non-blocking assignments, leading to race conditions (the only concurrent problem that can happen when two threads

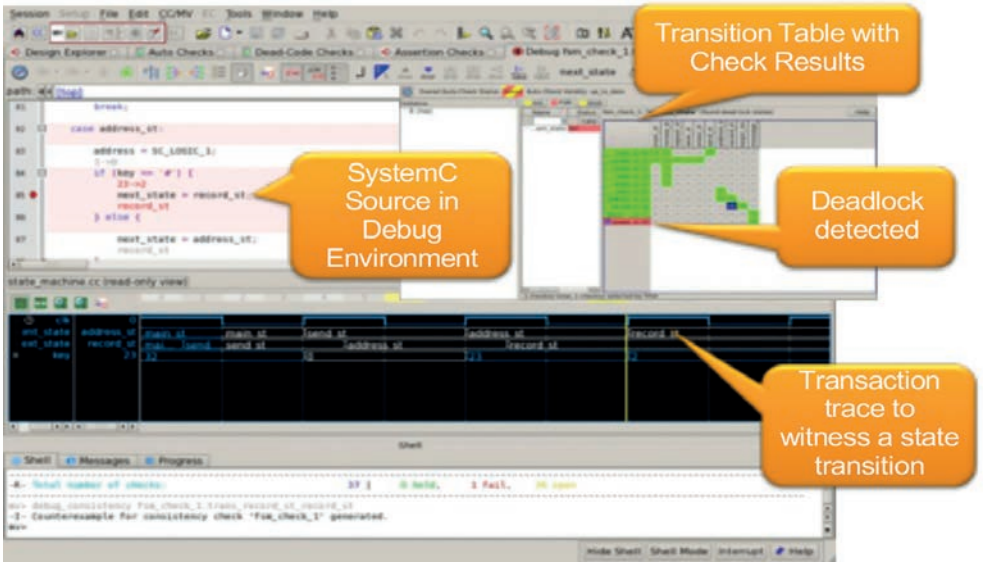

Figure 3: Deadlock checks in SystemC/C++ code

manipulate the same state, or value, in the same time lapse) and mismatches between sequential simulation semantics and parallel operation in hardware.

Many of the issues caught by DV-Inspect are not checked by simulation or HLS. These include unintended behaviour due to issues with specific data types, such as for fixed-point arithmetic, as well as concurrency-related issues such as race condition evaluation. Formal analysis provides a valuable pre-synthesis signoff to save overall development time and resources.

Sequential assertion-based SystemC/C++ verification

A full assertion-based verification solution for SystemC and other SystemC/C++ designs can be applied. Accepting the majority of SystemC functions, the formal solution allows assertions to be tested against a range of code abstractions, from transaction level models (TLMs) through detailed RTL and right down to netlists, and from almost untimed to full cycle accurate representations.

Both simple ANSI C assertions and fully temporal concurrent SystemVerilog assertions (SVA) may be used with the SystemC/C++ designs. This assertion description flexibility allows existing assertions for other designs to be reused or leveraged as templates, and reduces the learning overhead associated with a new format. This also enables consistent pre- and post-synthesis flow where the same assertions, if written with the flow in mind, may be reused on SystemC/C++ golden models and their RTL synthesised derivatives. In addition, verification intellectual property (VIP) assertion sets created for RTL environments, for example a bus protocol verifier, may be reused on SystemC/C++ code. This unique formal capability allows sequential assertions, which can be used to describe specification elements, expected design characteristics and fault conditions to be tested against abstract code. This allows engineers to work at the SystemC/C++ level on their golden designs to ensure that they meet their specifications prior to synthesis. It enables a comprehensive formal solution at a level where specifications may be played against different microarchitecture options. Finally, it eliminates the indirection of debugging a SystemC/C++ design using the post-synthesised RTL code.

Formal techniques

Formal techniques are well established as a key part of functional verification for hardware designs. Many designers have moved to SystemC/C++ to raise the abstraction level and take advantage of high-level synthesis. This approach speeds up the hardware design process but, for a corresponding reduction in verification time, the focus must be on the SystemC/C++ source code and not the post-HLS RTL design. The OneSpin DV solution for SystemC/C++ satisfies this need, providing both automated design inspection and full assertion-based verification for high-level designs. HLS users can take full advantage of the most advanced formal verification methods. EW

Structure (Easy Lint)	Safety Checks (Assertion Synthesis)			Activation (Coverage)
Mismatch/port /wire	Runtime Errors	Sim-Synth Issue	Safe Function	Dead code checks
Signal trunc / no sink	Array index	Initialization	Arithmetic overflow	Stuck signal (toggle test)
Sensitivity list issues	Function without return	X-Propagation	Redundant bits	FSM trans and states
Unused signal / param	Division by 0	Write-write race	Arithmetic shifts	MORE...

Figure 4: Wide range of checks for SystemC/C++ code