Fujitsu Network Technologies Ltd has accomplished two successful projects using the SystemC verification library (SCV) with Cadence Design System’s incisive unified simulator (IUS). Fujitsu’s team of engineers chose the IUS in creating an integrated environment because it has the ability to do multilingual simulation using SystemC and mixed hardware description languages (HDLs). The single environment allowed both design and verification engineers to seamlessly share the test and module IP, thus eliminating any rework.

Although the flow initially worked, the team soon realized its problems. For one, the C module was not specifically being reused in either the design or test of the RTL implementation. Moreover, the design was being developed and verified twice, once at the C level and once at the relatively slower RTL. Bugs were introduced into the RTL implementation because of misinterpretations by the RTL developer, typically when introducing time and cycle accuracy at the detailed signal-level implementation. With the advent of SystemC and its associated utilities and methodologies, much of these inefficiencies could be eliminated.

The new flow heavily utilized the notion of reuse to cut overall design and verification time by spending more time upfront, developing and verifying a model that can be reused throughout the RTL implementation phase. This provided the RTL developers with an executable spec and reusable testbenches, thus eliminating any rework.

Figure 1 compares two high-level project development flows: architectural exploration and a detailed cycle-accurate RTL design and functional verification.

Difficult to reuse due to the larger gaps with the differences of abstraction and language.

C/C++ language

Accuracy trade-off

RTL modeling

Verification

Previous development flow

(6 months)

C++/SystemC

New development flow

(4 months)

Accuracy trade-off

Abstract + function model

Behavior modeling

Verification

Reusing the function model and taking advantage of the behavior synthesis tool.

About 1/3 reduction

Much faster simulation against RTL.

Figure 1: SystemC is used to reduce development effort. The diagram compares two high-level project development flows: architectural exploration and a detailed cycle-accurate RTL design and functional verification.
Overall project time by 25%.

RTL design and verification interpretation. Less time in nating possibilities of any mis-

mance. Once the architects functionality and perfor-

rithm started at a very high level of abstraction. Concepts were first tried out using float-

level of abstraction. Concepts of HDL and verify perfor-

mance against highly abstract models. As the models were now verified efficiently to enable cal decisions no longer had to sign team. This stage of the process traditionally had many problems due to the gaps in abstraction between C/C++ and HDL. It was a rich source of functional bugs in the design. Thus, it was highly suggested to spend extra time at this stage to introduce timing and bit-accurate performance of HDL and verify against highly abstract system-level models.

Appropriate functionality at each level of abstraction was verified efficiently to enable architectural/structural decisions. As the models were now reused throughout the design process and into the RTL development phase, these critical decisions no longer had to be redeveloped by the RTL designer. By using the SystemC-based environment, RTL can be directly verified at each level of abstraction against reference models with the same intent and approximations as those used by system architects in the algorithmic and microarchitecture development phase.

For functional verification, the testbench was reused throughout the entire process with only transactor-interface modules needing enhancement to accommodate the increase in modeling complexity. Again, reuse of the testbench modules between SystemC and RTL greatly reduced the need for redevelopment and ensured design intent was maintained throughout the project cycle, while minimizing redundancy in the overall development effort.

The two case studies are isolated examples developed by the team for this design using advanced SystemC/SCV features:

Transactors as timing interface
The first project involved the design under verification written in Verilog and other parts of the testbench in SystemC/SCV. Transaction-level models (TLMs) of the CPU and RAM were written in SystemC. This section describes how transactors served as timing interface between RTL and TLMs and how CPU test scenarios were generated and simulated—including testbench random constraints for concurrent and random DMA sequences. RAM acknowledge returned with random delay across multiple clock domains, and data load and dump with randomly programed delays.

A graphical view of the test

Figure 3: This graphical view of the test plan reveals the test goals and analysis output.

Figure 4: Verilog DMA invocation sequence (pseudo-code) describes a function in the testbench initializing the CPU module to drive data to and from the DUV (DMA), forcing an interrupt in the DMA.

```c
void TestBench::task_dma0(int src, int dest, int size) {
    cpu.write(0x00000000, src); // Set Source address for DMA
    cpu.write(0x00000000, dest); // Set Destination address for DMA
    cpu.write(0x00000008, size); // Set transferring data size
    cpu.write(0x00000000, 0x0001); // Invoke DMA

    // Observation Interrupt
    while(true) {
        // detect interruption
        if (irq_i.read() == Log_1) {
            data = cpu.read(0x00000010); // Read status register
            irq_flag = data[0];
        }
    }

    // Clear Status register
    if (irq_flag == Log_1) {
        cpu.write(0x00000010, 0x0001); // Invoke DMA
        cpu.write(0x0000004, dest); // Set Destination address for DMA
        cpu.write(0x0000000, src); // Set Source address for DMA
    }
}
```

Figure 5: SystemC verification constraints are set for a desired traffic pattern to be generated.

const unsigned SRC_MIN = 0x55555555; // Source address [MIN]
const unsigned SRC_MAX = 0xFFFFFFFF; // Source address [MAX]
const unsigned DST_MIN = 0x33333333; // Destination address [MIN]
const unsigned DST_MAX = 0xFFFFFFFF; // Destination address [MAX]
const unsigned SIZE_MIN = 0; // Transferring size [MIN]
const unsigned SIZE_MAX = 64; // Transferring size [MIN]
```

// Declare smart pointer
scv_smart_ptr<unsigned> sizeP; // Transferring size
scv_smart_ptr<unsigned> srcP; // Source address
scv_smart_ptr<unsigned> dstP; // Destination address
scv_smart_ptr<unsigned> intervalP; // Transferring size
```

// Constraint
snP->keep_only(SRC_MIN, SRC_MAX);
dstP->keep_only(DST_MIN, DST_MAX);
sizeP->keep_only(SIZE_MIN, SIZE_MAX);
intervalP->keep_only(0,10);
```
plan (Figure 3) reveals the following test goals and analysis output:

1. Randomly-generated CPU commands to device under test for DMA transfers, random number of DMA transfers based on specified constraints—Maximum of eight transfers.

2. Acknowledge from RAM generated with random delay based on programmable constraints—Acknowledge with random time (0-30clk).

3. Dump the RAM data based on CPU commands.

4. Create transactions for analysis—All DMA traffic (Figure 8).

The test plan calls for a random set of CPU commands—writing and reading data to and from the DMA. SCV randomization is used to invoke the CPU/DMA transactions. Randomization provides a more realistic set of test data scenarios exercising the DMA and its registers.

The task of actually driving the DMA is developed as a callable function. The pseudo-code in Figure 4 is describing a function in the testbench initializing the CPU module to drive data to and from the DUV (DMA), forcing an interrupt in the DMA.

For the set of randomized data to be meaningful, they need to be constrained to remain within the test plan objectives and design parameters. Figure 5 shows how constraints were set for the desired traffic pattern to be generated—source/destination addresses, packet sizes and random REQ/ACK intervals were all constrained to remain within the specified ranges using smart-pointers and SCV constraints.

Figure 6 shows the outcome of the above. In reviewing the anticipated execution sequence, we can see that not only were the DMA write/reads happening with the random data and intervals, they were also being issued in parallel threads, exercising the DMA’s ability to handle multiple concurrent requests.

The team subsequently implemented transaction recordings such that the anticipated sequences could be captured and easily visualized in...
Transmission/FCS/HCS error Probability (1/1000)
Frame gaps range (1-150CLK)

Four types of frames with randomization

Variable length with constraint randomization
Variable length without constraints

Type1  Type2  Type4  Type4  Type3
Header   Data

Figure 9: For each frame generated, SCV randomized four elements of a frame packet.

the waveform viewer when simulating in the IUS. Using SystemC/SCV transaction-recording features and embedded SDI (transaction recording) calls within the SCV, the resulting simulations showed that the anticipated test sequences were executing correctly (Figure 7).

Transactions in the IUS platform are a simple and efficient way to record cause-and-effect sequences during a simulation run. When a problem in simulation is detected by the verification engineer, it is just a matter of reviewing the associated transactions to trace back to the origin of the problem. Any existing relationship between transactions—e.g. child-transactions spawned from a parent-transaction or error-transactions—can be analyzed using the explorer tool.

Dual frame generator TLMs

The second case study describes a test environment consisting of two frame generator TLMs that send random traffic to a VHDL design under verification. Engineers felt the necessity to somewhat stress-test the randomization capabilities of SCV from within VHDL before committing to full-scale test development. Similar to the first project, the randomization capabilities of SCV were used to create hierarchical constraints and generate random test parameters with the following objectives: random frame length and data; random frame gaps; and random error packets based on defined distribution.

Frame data was submitted upstream and compared downstream (Figure 8). Note that the SystemC verification modules were wrapped in VHDL to provide seamless signal-level integration with the DUV. For each frame generated, SCV randomizes four elements of a frame packet, creating a realistic scenario. Randomizing the header, data, source/test addresses and frame intervals enabled the frames to produce not only realistic traffic, but also error packets within a desired probability of 1 in 1,000 frames (Figure 9).

The team has concluded that SystemC can provide them with performance and flexibility to model algorithms to hardware, all within the same environment. Since the language is suitable for multiple design and verification tasks, they were able to implement their code for reuse throughout the design cycle. For debugging designs, transaction-level verification provided higher levels of performance, visibility and cycle.