Evaluation of a Refinement-Driven SystemC™-Based Design Flow

Thorsten Schubert[1], Jürgen Hanisch[2], Joachim Gerlach[2], Jens-E. Appell[1], Wolfgang Nebel[1]

[1]OFFIS Research Institute
Escherweg 2, 26121 Oldenburg, Germany
{thorsten.schubert, jens.appell, wolfgang.nebel}@offis.de

[2]Robert Bosch GmbH, Automotive Electronics
Tübinger Str 123, 72762 Reutlingen, Germany
{juergen.hanisch , joachim.gerlach}@de.bosch.com

Abstract

This paper describes the experiences and results that were made with a SystemC-based design flow for the implementation of an automotive digital hardware design. We present the refinement process starting from an initial high-level executable specification in C++ via SystemC down to a Gate-level description. We compare the synthesis results of the SystemC-based system-level design flow with those from a traditional VHDL-based register-transfer level design flow in terms of efficiency and simulation performance.

1. Motivation

Today's design flows are characterised by a heterogeneous mixture of design languages and tools. Different domains like analog hardware, digital hardware and software each have their specialised languages and tools. Even when restricting to a single domain like digital hardware, still a diversity of languages and tools is used. For example, Matlab is very popular for algorithmic modelling, especially in the signal processing community. C and C++ are often used for similar purposes when high simulation performance is required. For the actual implementation of hardware, RTL modelling with VHDL and Verilog is predominant.

These language changes between the different stages of the design process require manual recoding, which is laborious and error-prone. Furthermore, language changes in the design flow complicate the verification of the design. Either a co-simulation between the different languages has to be done, or the testbench has to be rewritten in the new language.

SystemC promises to overcome this problem by providing a single modelling framework that covers a wide range of abstraction levels. For example the concept of hierarchical channels facilitates transaction-level modelling (TLM), i.e. an abstract way to model communication in order to explore and profile architectural alternatives [1]. For describing algorithmic designs, SystemC introduces the notion of untimed and timed functional abstraction levels. On these levels of abstraction the designer can concentrate on the functionality of the design, while abstracting from details of communication and synchronisation. Furthermore, SystemC is able to describe systems at the Behavioural- and RT-level. While the higher levels of abstraction may be used for simulation purposes only, it is possible to use Behavioural-and RT-level descriptions as starting point for an automated synthesis very similar to VHDL and Verilog.

A further promoted advantage of SystemC is its high simulation performance, which is achieved by a compiled execution and the use of higher levels of abstraction.

In order to be applicable in an industrial design flow, it is mandatory that the new approach seamlessly integrates into existing design flows. Regarding our design example from the digital hardware domain, this means we have to build on top of the well established and mature HDL-based RTL flow in terms of synthesis and simulation. Tools supporting the new design flow must be commercially available and supported.

The second major prerequisite is the efficiency of the synthesis results in terms of area and timing.

Regarding verification, it is always desirable to obtain a high simulation performance in order to achieve a sufficient certainty of functional correctness.

The rest of this paper is organised as follows. Section 2 gives an overview of the evaluation procedure. Section 3 describes the design example that is the basis of our evaluation. In Section 4 we describe the refinement process. We present experimental results in Section 5 and draw a conclusion of the evaluation in Section 6.

2. Evaluation Procedure

In order to evaluate the applicability of SystemC and its potential advantages, we chose a design flow as depicted in Figure 1. A stepwise manual refinement was applied starting with an initial specification in C++ and ending with two different kinds of synthesisable SystemC models: a behavioural and a RTL model both complying with the corresponding synthesisable subset of the SystemC Compiler from Synopsys®. Each refinement step was verified for bit accuracy by simulation. Both models were optimised in order to improve the synthesis results. The optimisation goal was a minimum area under a fixed timing constraint of 40 ns (clock period).

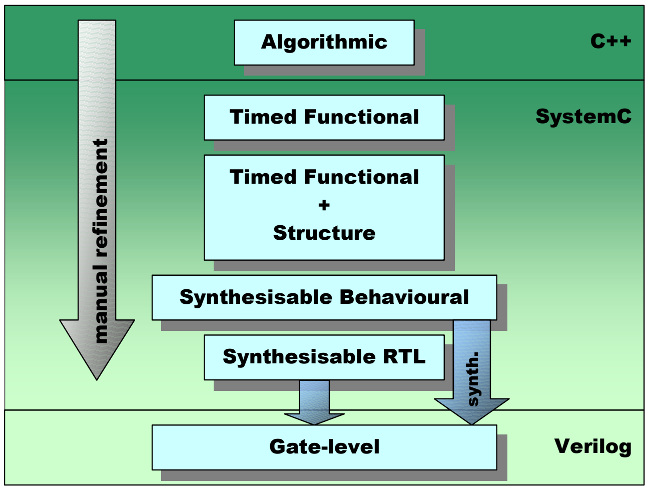

Figure 1. Design flow

C++ was chosen over Matlab for the initial specification in order to be able to stay in a single language environment as long as possible during the design process.

An existing, series-production quality VHDL implementation of our design example, that was created with the conventional flow of manually recoding the given C specification in RTL VHDL served as reference implementation regarding the required efficiency.

3. Design Example

The design example we chose for our evaluation is a sample-rate converter (SRC). It represents a typical hardware design in the area of car multimedia. Its moderate complexity (about 3000 lines of code for the final RTL-SystemC implementation) makes it an ideal candidate for an evaluation. The purpose of the SRC is to convert stereo audio signals between different sampling frequencies from different sources, e.g. between 44.1 kHz (CD) and 48 kHz (DVD). This is illustrated in Figure 2. Given a sequence of samples of an analog signal equally spaced at T_{In}, it is SRC's task to calculate samples of the original analog signal at a different rate. So basically it is an interpolation problem. The underlying algorithm that realises the interpolation makes use of bandlimited interpolation as described in [1].

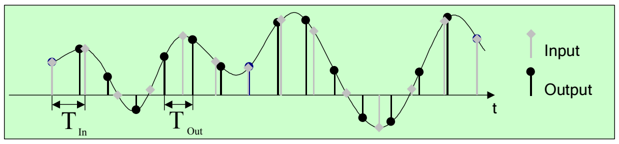

Figure 2. Sample-rate conversion

The SRC can be regarded as a periodically time-variant system, i.e. it performs a convolution of the input signal with a time-varying impulse response. In order to do so all SRC implementations from our evaluation contain the following parts: a buffer, that collects past input samples, some kind of ROM to store the impulse responses (filter coefficients), and an algorithmic block that performs the convolution (see Figure 3 - Figure 6)

4. Refinement Process

The overall strategy during refinement was to make only small, conservative changes to the design and to revalidate each refined model. In this section we describe the major refinement steps.

4.1. Initial Specification in C++

The basic structure of the C++ version is shown in Figure 3. The three main components are the classes `CInputBuffer`, `CPolyphaseFilter` and the function `Filter()`.

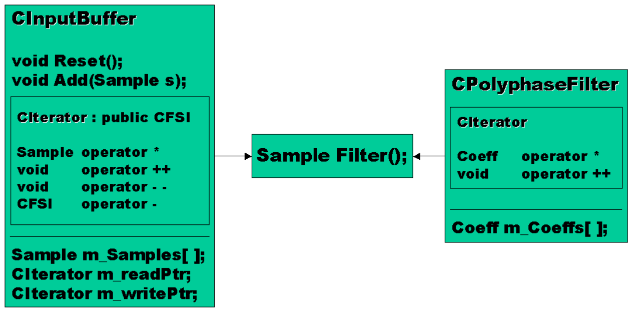

Figure 3. C++ model of the SRC

The class `CPolyPhaseFilter` (more precisely: an instance) handles the storage of the coefficients for the time-varying impulse response. The actual filtering is done in the function `Filter()`. With each call to `Filter()` one output sample is calculated. One might expect to find this as a member function of the class `CPolyphaseFilter`, but the filter needs the samples from the input buffer in the same way it needs the coefficients of the polyphase filter. Consequently the filter function was associated to neither of the classes. The filter function is realised as a single function, which obtains the samples as well as the coefficients in the same way, namely via the iterators provided by the input-buffer and the polyphase-filter class (Figure 3). The iterators can be regarded as some kind of access objects similar to pointers. They provide methods to access the element they are pointing on and to manipulate the pointer. The iterators for the input buffer realise a ring buffer like access scheme. They can be thought of as read and write pointers (see Figure 4). The iterator internally holds an index to an array and ensures a correct wrap around, because it can only be modified through public methods. For example when the convolution steps backwards through the input samples, the iterator automatically wraps from 0 to the maximum index. In a similar fashion the iterator of the polyphase filter hides the storage order of the coefficients and the fact that only one half of the symmetrical impulse response is stored.

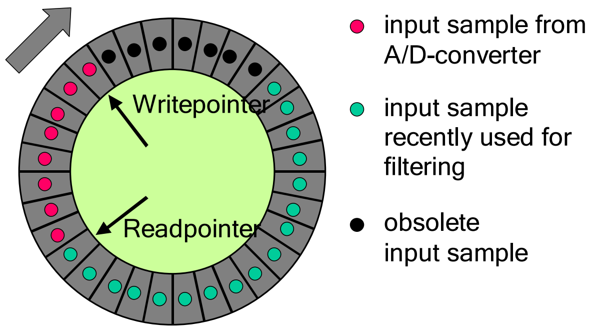
Writepointer
Readpointer
input sample from A/D-converter
input sample recently used for filtering
obsolete input sample

Figure 4. Input buffer organised as ring buffer

4.2. SystemC 2.0 with Channels

Starting from the C++ implementation as the golden model, an abstract SystemC model was developed. The first refinement step was a structural refinement, which resulted in the model shown in Figure 5. The SRC algorithm was encapsulated in a hierarchical channel, which implements the three interfaces `SRC_CTRL`, `SampleWriteIF` and `SampleReadIF`. The `SRC_CTRL` is the configuration port for setting the operation mode.

The major difference between the C++ and the SystemC implementation is the way the model is executed in its testbench. The C++ model is a pure sequential algorithm. The calculation of an output value is assumed to be performed in zero time. The time between two output samples is calculated and the number of input samples that would have arrived during this time is added to the input buffer, before the output sample is calculated. The SystemC model behaves differently. The producer and consumer in Figure 5 and Figure 6 are independent threads which read and write samples with a certain frequency.

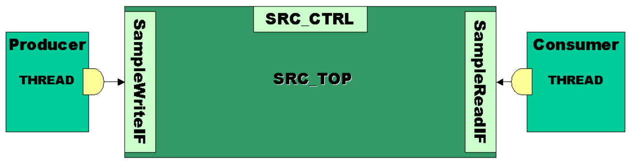

Figure 5. SRC as hierarchical channel

In a next refinement step the hierarchical channel itself was refined. The C++ code was split-up into three submodules, basically according to the class structure. A third thread was added in the main module of the SRC modelling its functional behaviour. Synchronisation between the threads was done by explicit event objects (`sc_event`). The method calls of the C++ model were roughly translated into interface method calls (IMC) through channels [3].

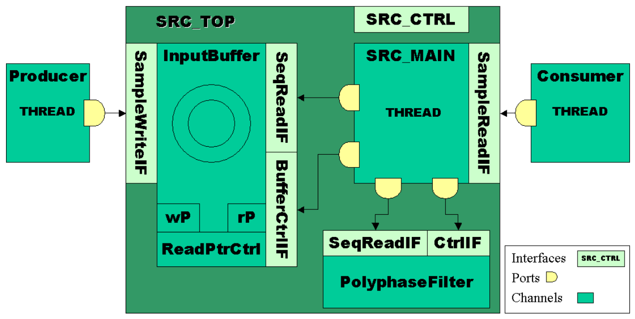

Figure 6. Refined hierarchical channel

4.3. Synthesisable Behavioural SystemC

The refinement of the non-synthesisable SystemC implementation with channels into a synthesisable behavioural description required the following steps:

Communication refinement: The IMC-based communication of the hierarchical channels was replaced by a signal-based communication. Explicit handshaking was implemented, because the main module was going to be scheduled in a scheduling mode where the number of clock cycles between I/O operations is not fixed [4]. Therefore handshaking signals have to be used to indicate valid data. The advantage of this scheduling mode is that it offers the greatest optimisation potential.

Structural refinement: All arithmetic operations were moved into a single process allowing resource sharing for more efficient synthesis results with current synthesis technology.

Type refinement: The native C/C++ types were replaced by SystemC types with explicit bit-widths. Although the native types would have been synthesisable, the use of explicit bit-widths produces more efficient synthesis results.

Timing refinement: A clock was introduced. The time quantisation that was introduced by this clock required a change of the golden model. The effect is illustrated in

Figure 7. The upper part of the figure shows the sampling times within a continuous time domain; the lower figure shows the sampling times as "seen" by the clocked implementation (discrete time domain). Since the events at which input and output samples occur can only be detected at clock edges, these events are slightly delayed. This delay causes small changes in the output values compared to the reference data. To be able to still compare the output values with the reference output values by a bit-accurate comparison, the time quantisation was manually propagated back to the golden model.

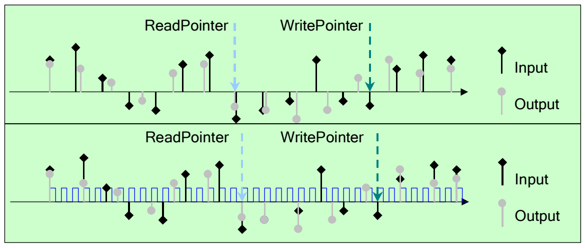

Figure 7. Time quantisation of sample events

Note, this behavioural version of the SRC already contained RTL SystemC modules for two reasons. The first reason was to test the interoperability of behavioural and RT level code for simulation and synthesis. The second reason was that some modules (especially the I/O interfaces) only contained simple control functionality, which was easy to implement at RTL.

4.4. Optimised Behavioural SystemC

The first synthesis result of the complete SRC needed 27.5% more area than the VHDL reference implementation (for the results refer to Section 5). By far the biggest part of the design with more than 90% of the total area was the `SRC_MAIN` module. Therefore the optimisation was focussed on this module.

The following constructs were expected to cause inefficiencies of the synthesis result:

Handshaking in loops. A handshaking mechanism between the input buffer and the `SRC_MAIN` module was necessary, due to the scheduling policy of the behavioural synthesis. The handshaking mechanism could be eliminated by relying on a fixed cycle scheme. For example this can be done by proper constraining the behavioural code.

Code proliferation. Because the focus of the refinement process was to preserve the functionality a conservative "cut-and-paste-and-refine" strategy was chosen. Since major restructuring of the code was necessary the code became cluttered and less readable. An intensive code cleanup at this stage helped to simplify several expressions.

Bit-widths. Due to the conservative refinement strategy, some bit-widths were chosen too pessimistic. These could be reduced without affecting the result.

Generality. The initial C++ implementation of the SRC was written in a very generic style. Especially the bit-widths were parameterised through the C++ template mechanism. The template mechanism was replaced by preprocessor `#define` directives. The code was more flexible with parameterised bit-widths, but it was harder to figure out which constructs caused large registers, operations, etc.

4.5. RTL SystemC

The manual refinement of the synthesisable behavioural SystemC code from the main module into synthesisable RTL code consisted of the following steps:

- Fine-tuning of the model's scheduling.
- Allocation of registers for the variables
- Creating an FSM that realises the scheduling

The data-path was not modelled explicitly. Instead it was described implicitly by the state transitions of the FSM and then optimised with the Design Compiler™ [5].

The refinement of the behavioural SystemC implementation to an RTL description was relatively easy. The optimised behavioural SystemC implementation of the SRC was already bit-accurate and nearly cycle-accurate, so the main task was to optimise its scheduling and the creation of the controlling FSM.

Although RTL-modelling and synthesis is not the primary area of application for SystemC and is less elegant than with HDLs, it is definitely feasible.

4.6. Optimised RTL SystemC

Since the data-path already utilised a minimum number of resources, the remaining optimisation potential results from register usage. Since the refinement was done in a conservative way, there were still some registers that could be eliminated.

4.7. Discussion

The first refinement steps turned out to be the most difficult and labour intensive tasks, because of the semantic gap between the different levels of abstraction, e.g., the transition from a sequential C++-based program into concurrent SystemC processes and their transition into clocked threads. In total, nearly the same amount of work as in the case of recoding the model in VHDL had to be done, but in a stepwise manner instead of a single large transformation. If too many, intermediate levels of abstraction and refinement steps are chosen, the effort for the refinement can be even higher. Regarding this evaluation we think that the use of channels was inadequate for our application. Channels neither helped for profiling or exploring different architectures nor did they help for synthesis.

The refinement effort however, is comparable to the recoding effort, at least when targeting the RT-level. This appears sensible due to the fact that the necessary tasks, e.g. creating an architecture and scheduling, are the same – just the language is different.

An advantage of the refinement-driven approach is that parts of the code can be reused and that the risk of introducing new errors into the model is lower. During our evaluation it even happened that a bug in the golden model was refined down to Gate-level and was discovered during Gate-level simulation. The bug in the golden model has been identified as an erroneous access to an invalid buffer position in some corner cases. When the memory for the buffer was replaced by an automatically generated simulation model (that included a check for valid addresses) for Gate-level simulation, the bug became obvious. On the one hand this example shows that the (in principle positive) function-preserving property of the refinement-driven approach also has some drawbacks. On the other hand the example shows, that the refinement approach allows for testing the system's general functionality across all design-stages, so that the probability for failure recognition is increased in this approach

The synthesis results in Section 5 show that the most efficient designs could be obtained with the RTL-SystemC implementation. The effort to refine the optimised behavioural level description into RTL turned out to be lower than expected. So at least for this kind of design the RTL approach has the best cost-value ratio. The feature making behavioural synthesis less suitable for this class of designs is the asynchronous nature of sample-rate conversion in general. It requires a relatively fine-grained control over the internal timing behaviour, a quality that is much easier to control at the RT-level.

5. Results

5.1. Simulation Performance

All simulations have been done on a Sun Blade 100 with 500 MHz and 640 MB RAM. All simulations were performed with ModelSim® 5.5d, SystemC models were compiled with gcc 2.95.1 and SystemC/HDL co-simulation has been performed with the SystemC HDL-Cosim tool version 2002.05 from Synopsys (which is now part of System Studio). The simulation performance is given in simulated clock cycles/second. The implementations without a clock were scaled appropriately according to the ratio of simulation time and simulated time assuming a 25 MHz clock.

Figure 8 shows the degradation of the simulation performance along the refinement process. As expected, the pure C++ implementation is the fastest one. When comparing the simulation performance of the behavioural SystemC and RTL SystemC implementation, it has to be considered, that the behavioural level implementation also contained RT level components. Due to the lack of proper profiling tools for the SystemC simulation, it could not be checked whether the RTL parts dominated the overall simulation or whether the behavioural part is not significantly faster at all.

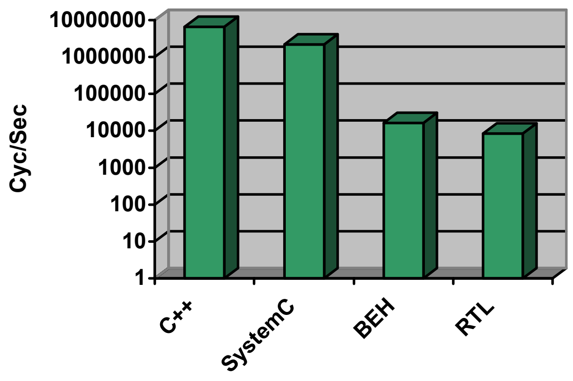

Figure 8. Simulation performance on different levels of abstraction

Figure 9 shows the simulation performance of the HDL versions of the SRC: the intermediate RTL Verilog code from RTL SystemC synthesis, the Gate-level code (Verilog) resulting from the behavioural flow and the Gate-level (Verilog) code resulting from the RTL flow. The simulations have been performed in two different configurations. In the first configuration each design under test (DUT) was simulated in the VHDL testbench, that was available from the reference design. In the second configuration each DUT was simulated in the SystemC testbench.

As can be seen from the figure, the co-simulation of the DUT in the SystemC testbench is slightly faster than a native HDL simulation. This indicates that in this case the performance gain by using SystemC outweighs the overhead introduced by the co-simulation.

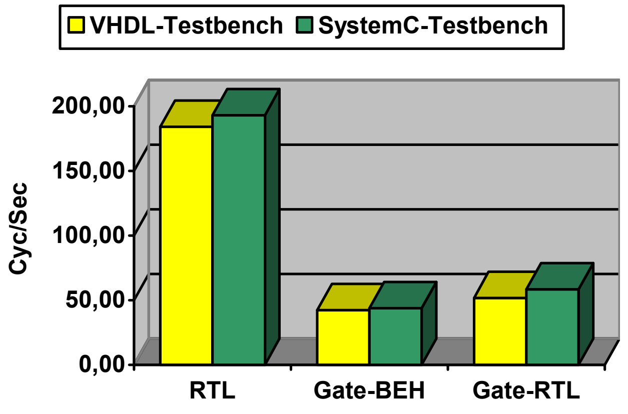

Figure 9. Co-Simulation vs. native HDL simulation

5.2. Synthesis

Synthesis has been done with SystemC™ Compiler 2002.05 and Design Compiler™ 2002.05 from Synopsys. Target library was a 0.25µ CMOS library. All designs were synthesised with the same synthesis constraints.

We only consider the area here, since it was the main optimisation goal in the evaluation. The timing goal could be easily achieved by all implementations.

Figure 10 shows the area of the SRC designs after compilation to Gate-level relative to the VHDL reference design which is scaled to 100%. The area numbers were obtained by the `report_area` command of the Design Compiler. Memories are excluded from the area, because they are identical for all implementations and do not reflect the quality of the synthesis result. A scan chain, however, is included in all designs.

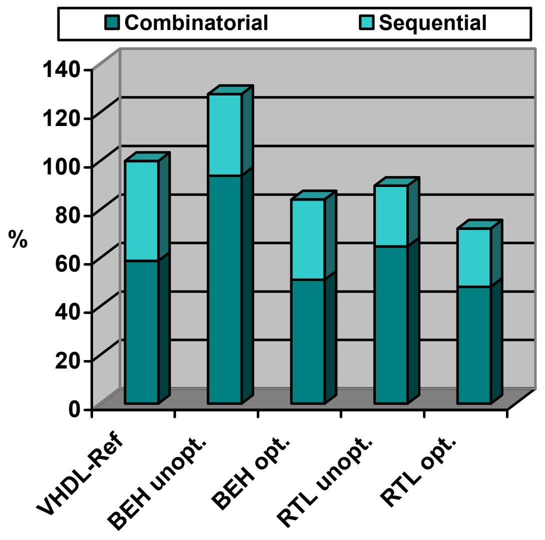

Figure 10. Comparison of area efficiency

The most remarkable result is that the optimised SystemC implementations produce a smaller design than the VHDL reference. Even the unoptimised RTL-SystemC implementation is smaller than the VHDL reference. We can explain this result as follows. The VHDL reference implementation started from a low level C specification that already guided the implementation to a specific architecture. The more abstract C++ model includes more degrees of freedom concerning the target architecture. So actually the efficiency was gained through the use of more abstract models. A direct refinement of the original low-level specification into RTL-SystemC would have resulted most probably into nearly the same result as the VHDL implementation.

When comparing the optimised behavioural and RTL implementation it can be seen, that the amount of combinatorial logic is nearly the same. This indicates, that the optimum allocation was reached with the behavioural synthesis. The area savings of the RTL SystemC implementation over the behavioural SystemC implementation result from a more efficient usage of registers.

6. Conclusion

Our evaluation showed that it is generally possible to apply SystemC in an industrial design flow. The integration into the existing design flow turned out to be straight forward.

Regarding the efficiency of the synthesis result our evaluation showed that the refinement-driven approach and the use of higher levels of abstraction does not necessarily produce less efficient results. Even the contrary was the case in our evaluation.

As expected, the use of higher levels of abstraction allows for much faster simulation. Even co-simulating the SystemC testbench with the HDL design turned out to be slightly faster than the pure HDL simulation.

The refinement-driven approach in a single language has pros and cons. The main advantage of the approach is the partitioning of the design process in several intermediate steps combined with a revalidation of each step, which allows for making small function-preserving changes. The disadvantage is that undersized refinement steps will result in an overall higher effort than complete recoding and tend to produce lower quality code, when refinement means changing the code instead of completely rewriting it. With a careful selection of the right abstraction levels and the ongoing raise of the level of automation of the SystemC synthesis process, the advantages could outweigh the disadvantages.

7. Acknowledgements

This work was done in the MEDEA+ project SpeAC and partially supported by the German Bundesministerium für Bildung und Forschung under grant 01M 3049C.

8. References

[1] Grötker, T., Liao, S., Martin, G., Swan, S., *System Design with SystemC*, Kluwer, Academic Publishers, Boston, 2002

[2] Digital Audio Resampling Home Page, http://www-ccrma.stanford.edu/~jos/resample/

[3] Open SystemC Initiative (OSCI), *Functional Specification for SystemC 2.0*, October 2001

[4] Synopsys Inc., *CoCentric® SystemC™ Compiler Behavioral User and Modeling Guide Version 2002.05*, June 2002

[5] Synopsys Inc.: *Design Compiler™ Reference Manual: Optimization and Timing Analysis, Version 2002.05*, June 2002