AN EXTENSION TO SYSTEMC TO ALLOW MODELLING OF ANALOGUE AND MIXED SIGNAL SYSTEMS AT DIFFERENT ABSTRACTION LEVELS

H J Al-Junaid and T J Kazmierski

University of Southampton

Abstract

SystemC is Hardware Description Language HDL for digital systems. An extension is proposed in this paper to extended the capabilities of SystemC to allow modelling of analogue and mixed-signal systems. The proposed extension provides a variety of abstraction levels, from system level to circuit level. In order to comply with the SystemC simulation cycle semantics, the analogue kernel is linked to the SystemC environment via calls from the existing digital kernel. The synchronisation of the analogue and SystemC digital kernels is done via a lock-step method. Operation of the extended, mixed-signal SystemC simulation platform is demonstrated using a practical example of a phase locked loop frequency multiplier with noise and jitter. We hope that results from this research might aid the recent efforts to standardize analogue extensions to SystemC.

1 INTRODUCTION

The need to integrate complete complex systems on a single Chip SoC has started a new era in design automation. SoC has created a need for powerful CAD tools and methodologies which are capable of integrating information from multiple heterogenous sources (analogue parts, processors, RAM, ROM, etc.) and have the ability to work at high level of abstractions.

Furthermore, Analogue and Mixed-Signal (AMS) high-level modelling is lagging behind the digital design due to its immature design methodologies [1]. This created a gap in the design of the two different parts which slow the production rate. It is essential to include analogue components and a system environment into an overall simulation HDL like VHDL-AMS or Verilog-AMS which allow a description of mixed signal system.

The recent trend in digital system design is toward C++ based modelling [2] either through libraries like SystemC, Cyblib or OCAPI or through abstractions like SpecC. SystemC [3], one of the newest hardware description languages, has become the subject of growing interest throughout the electronic industry since the release of the first version in September 1999. SystemC is a standard modelling language intended to enable system level design and IP exchange at multiple abstraction levels for complex systems containing both software and hardware components.

There is an extensive research towards SystemC specification, co-simulation, co-design, co-verification and synthesis of systems at different abstraction levels. Until recently, there have been few research papers directed towards extending SystemC to modelling AMS systems.

For instance, Einwich et al [4] presented a framework support for signal processing dominated application. The framework is based on analogue extensions for DAEs (differential and algebraic equations) and frequency domain simulation. Linear DAE solvers are integrated into the synchronous data flow design. An AMS simulation framework is presented by Bonnerud et al [5] for simulation of analogue to digital data converters ADC. The framework contains a C++ mixed signal module library that includes a set of flexible and customizable primitives, compound modules and test-benches.

Another approach, proposed by Conti et al [6], allows a description of analogue systems at low or higher level using analogue macro-models; it adopts a threaded analogue modules system. The methodology was applied to a fuzzy controller and a CMOS inverter chain oscillator. Grimm et al [7] introduced an ASC library, a prototype for AMS extension to SystemC. The ASC library provides analogue or signal processing behavioural processes and their execution is controlled by a coordinator interface.

None of the papers listed above provides an approach with a general simulator for the analogue parts to solve non-linear systems with variable time step and addresses other essential issues which are necessary to model general AMS systems. The aim of this research is to extend SystemC to model analogue and mixed-signal systems at a variety of abstraction levels, and consequently, develop a general non-linear analogue simulator which works in parallel with the digital simulator and both interact at specific time points when needed. A particular attention has been devoted here to the problem of synchronising the analogue kernel with the SystemC digital kernel. Our synchronisation techniques are compliant

with the definition of SystemC simulation semantics [8].

A working group was formally established on February 2003 to develop an extension to SystemC called SystemC-AMS under the support of the Open SystemC initiative OSCI. In this respect, the results presented here might aid the recent efforts to standardize analogue extensions to SystemC.

In this paper, elements of the AMS extension which are addressed within the scope of this research are described in Section 2. Section 3 illustrates the implementation of the digital-analogue interfaces and handles some problems which arose when putting together the analogue and digital parts. Section 4 explains the SystemC simulation cycle and how it is linked and synchronised with our AMS extension. Finally, the proposed extension was verified by modelling several examples but Section 5 gives one case study of modelling a high-speed phase locked loop with noise and jitter, which is a non-trivial AMS system.

2 ELEMENTS OF THE AMS EXTENSION

The new classes added to language cover the most important aspects of AMS modelling. They include support for analogue System variables, analogue components, corresponding virtual build methods used by the underlying solver and the implementation of analogue to digital interfaces. A corresponding analogue kernel has been constructed which simulates a user code describing the system in a simple and familiar form such as a SPICE-like net-list or VHDL-AMS-like simultaneous equations.

2.1 Analogue System Variable

In order to provide a mechanism for modelling non-linear AMS systems, the new language extension should provide a notation for DAEs. In the set of DAEs Eq. 1, the analogue system variables introduced into the extension ($\mathbf{v}(t)$) represent the unknowns.

$$\mathbf{f}(\mathbf{v}(t), \dot{\mathbf{v}}(t), t) = \mathbf{0} \quad t \geq 0, \quad \mathbf{v}(0) = \mathbf{v}_0 \qquad (1)$$

The C++ concept of inheritance is used to define various types of analogue system variables, such as nodes, currents, free variables and others. In the proposed extension, they represent a hierarchy of system variables, all derived from an abstract base class as illustrated in Fig. 1. Currently only three types of variables derived from the base class have been defined, `sc_a_node`, `sc_a_flow` and `sc_a_free_variable`.

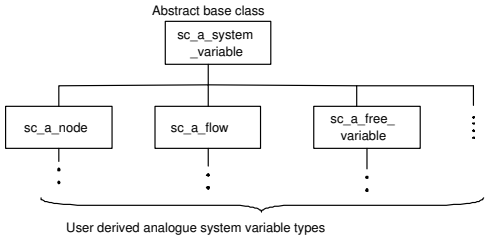

Figure 1: Analogue system variable inheritance hierarchy.

`sc_a_node` is used to represent node voltages in electrical circuits. Where `sc_a_flow` is used to represent flow variables (e.g. electric currents) in MNA-like equation formulations. According to the MNA representation of some components, like a voltage source or an inductor, a current variable should be introduced in conjunction with the declaration of any of these components. The free system variable `sc_a_free_variable` is introduced to define variables when describing a system or part of it by a differential equation rather than a networked circuit component. It is useful especially when modelling systems at behavioural level for describing the functionality of system blocks.

2.2 Analogue Components

Analogue circuit components have been proposed here to provide equations which describe analogue behaviour. Similarly to the system variable hierarchy, components are derived from an abstract base class which contains a virtual `build` method invoked by the analogue kernel. A sample component class hierarchy is illustrated in Fig. 2 with examples of SPICE-like circuit elements such as resistor, capacitor, inductor, diode and various types of autonomous sources. Arbitrary differential and algebraic equations can be included as user-defined components.

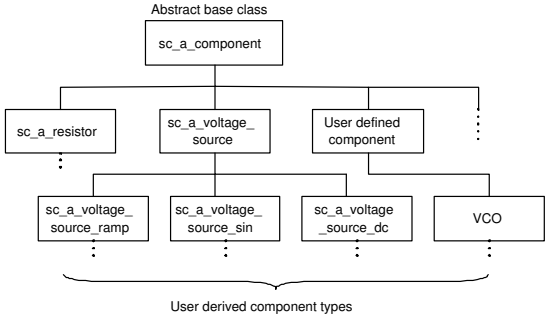

Figure 2: Inheritance of analogue components.

The typical component class would contain a pair of node pointers and a value. An example of instantiating a capacitor is shown below:

```
sc_a_capacitor *c1= new sc_a_capacitor("c1",
                                nodeA, nodeB, C)
```

where `sc_a_capacitor` is a new component class derived from the base abstract class, `c1` is the component name, `nodeA` and `nodeB` are names of analogue system variable objects of type `sc_a_node` and represent the two terminals to which the capacitor is connected, and `C` is the capacitance.

The base class constructor attaches each newly created component to a global linked list of system components to form a connected circuit. The list is used at the matrix build time by scanning all the components to invoke their `build` functions.

A net-list of an analogue circuit can be constructed by declaring system variables of type node and analogue components as shown below of the loop filter in a phase locked loop (explained later in Section 5). Fig 3 shows its corresponding schematic. The circuit's data base is constructed once, prior to a simulation.

```
sc_a_node n1("n1"), n2("n2"), n0("n0");
sc_a_currentS I1("I1",n1,n0,&Iin);
sc_a_capacitor c1("c1",n1,n2,3e-9);
sc_a_resistor r1("r1",n2,n0,1e3);
sc_a_capacitor c2("c2",n2,n0,4e-9);
```

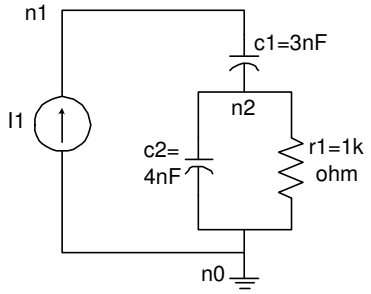

Figure 3: Schematic of the loop filter in PLL example.

2.3 Virtual Build Method

The `build` method specifies the analogue behaviour of a component. This is a virtual method with a default body in the abstract component base class and inherited by all derived components. The `build` method consists of C++ code which defines one or more DAEs. For example, Fig. 4 shows the capacitor representation. The figure shows the capacitor's differential equation, its representation after discretisation and part of the corresponding `build` method. The resulting Jacobian stamp conforms to the Modified Nodal Analysis formulation MNA. Calls to `BuildRhs`, build the differential equations for the capacitor or the right hand side RHS. Calls to `BuildM`, which build the corresponding Jacobian entries are optional. If these calls are not provided, the solver will build the Jacobian using a secant approach with finite difference approximation of the Jacobian entries. The entire equation set is formulated automatically at each Newton-Raphson iteration by scanning the linked list of components and invoking their build methods.

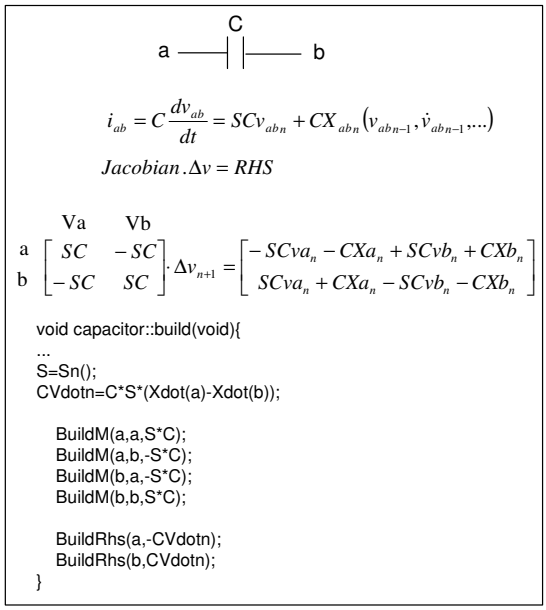

Figure 4: Capacitor equation and build function.

3 DIGITAL-ANALOGUE INTERACTION

Connectivity between analogue and digital models requires special consideration since the two models have different language representations. The solution to this problem is to insert a special interface model directly between the digital and analogue parts. The intended interfacing solution is similar to those adopted in VHDL-AMS and Verilog-AMS. A/D and D/A interfaces are used only to change representations of signals between the digital and analogue domains.

3.1 Digital-Analogue Interface

`interfaceDA` is a SystemC module which contains an input port of type *bool* and an output port of type *double*. `interfaceDA` ports are connected to signals of the corresponding types. A digital signal coming from digital module are transformed into analogue signal and directed towards the analogue module through the output port. Digital signal may introduce instability in the analogue simulation due to large instability changes in node voltage when the digital node switches. Therefore rather than changing abruptly, a transformation is done by a smoothing function. The smoothing is done by Eq. 2 and

shown in Fig. 5. This method is capable of handling small time step size as well.

$$S_n = \frac{S_n h_n + \tau S'_{n-1}}{\tau + h_n} \quad (2)$$

where S_n is the input digital signal of type *bool*. h_n is the simulation time step size. S'_n is the smoothed signal and S'_{n-1} is the past value of the smoothed signal. τ is time constant which plays as a control factor to shape the signal.

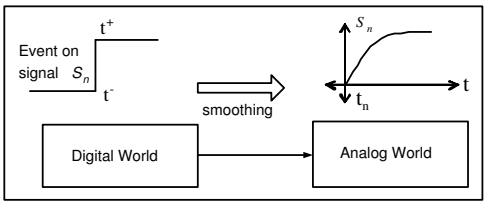

Figure 5: Handling small time step sizes.

3.2 Analogue-Digital Interface

`interfaceAD` is a SystemC module takes analogue signal of type *double* and produce a digital *bool* signal. The criteria to generate a digital event is simple, if the threshold voltage E defined is exceeded, an event with a state (high) is generated. An event with a state (low) is produced, if the analogue voltage falls below the threshold voltage. Due to the fact that the result is a digital boolean signal, an event is to be generated at every signal change. The digital part will react to this event if a concurrent statement reads this signal or if the sensitivity list of a process contains this signal.

3.3 Analogue Stepping

The time step of the analogue simulator is determined by the internal algorithm of the simulator, which means it cannot be defined by the user. Analogue simulators do not use events but instead employ an entirely different approach to time step control, namely, continuous step size adjustment, as illustrated in Fig. 6, where $h = h_n, h_{n+1}, ...$ may have different values.

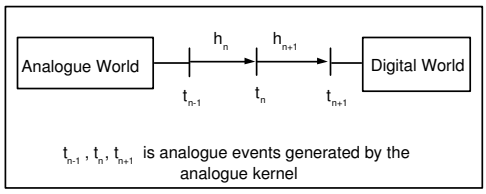

Figure 6: Analogue stepping.

The implementation of analogue stepping is based on the estimation of the Local Truncation Error LTE. LTE at t_n is an error due to a numerical approximation introduced in the time point t_n.

In order to synchronise the analogue and digital simulators at every time point, the analogue stepping is done in SystemC using *event* notifications. The analogue kernel which is responsible for calculating the estimated value of the upper step size bound h_n notifies the kernel at the time point equal to `sc_time_stamp()` $+h_n$. The digital processes will be activated at this time point accordingly.

4 TIME SYNCHRONISATION BETWEEN ANALOGUE AND DIGITAL SOLVERS

The most important problem in mixed-signal simulation is the time synchronisation between the event-driven digital simulation and the numerical integration in the analogue simulation. Synchronisation is a key issue affecting the simulation speed and accuracy. Illustrated in the following sections the SystemC simulation cycle and the way our analogue kernel is linked to it.

4.1 SystemC Simulation cycle

Like in the case of most high-level HDLs, a SystemC model consists of a hierarchical network of parallel processes, which exchange messages under the control of the simulation kernel process [3] and concurrently update the values of signals and variables. Signal assignment statements do not affect the target signals immediately, but the new values become effective in the next simulation cycle [8]. The kernel process resumes when all the user defined processes become suspended either by executing a *wait* statement or upon reaching the last process statement. On resumption, the kernel updates the signals and variable and suspends again while the user processes resume. If the time of the next earliest event t_n is equal to the current simulation time t_c, the user processes execute a delta cycle.

4.2 Proposed Mixed-Signal SystemC Simulation cycle

In this proposed research, the digital and analogue simulation cycles are combined. Hence, a set of computations of the analogue equations is executed between the digital evaluation points. To comply with the SystemC execution semantics, the proposed mixed-signal simulator comprises an analogue kernel (see Fig. 7), which runs as a SystemC process and drives the user defined analogue modules.

The analogue kernel repeatedly executes its simulation cycle, which might involve delta cycles and backtracking. Analogue simulators use continuous

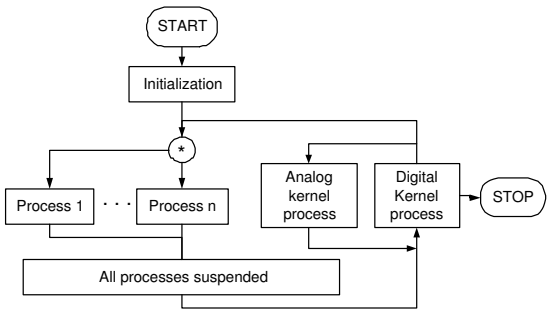

Figure 7: The proposed simulation cycle of a SystemC system with analogue kernel.

step size adjustment to minimize the errors caused by the numerical integration method.

It is therefore necessary for the analogue kernel in a SystemC environment to handle delta cycles in a manner similar to that of digital processes. However, the state of the analogue solver may not be updated until after the SystemC kernel advances the simulation time ahead of the current simulation time t_c, unless a delta cycle occurs and reevaluation of the current step is necessary.

The technique used in this project for synchronisation is the lock-step one. The analogue simulator calculates the step sizes and the digital simulator uses these values. The analogue kernel advances until the current simulation time and, before suspending, schedules an event at the time equal to the current simulation time plus the next selected step size. The method has been implemented in the extended language by modifying the SystemC kernel specified by (sc_simcontext.cpp) from the SystemC library.

5 CASE STUDY: 2GHZ PHASE LOCKED LOOP

To verify the functionality of the proposed SystemC mixed-signal simulator, a case study of modelling a 2GHz Phase Locked Loop (PLL) is illustrated. PLL is non-trivial system to model. Systems of this kind usually put standard SPICE-like simulators into difficulties because of the disparate time scales of their transients. As a typical simulation in a system of this kind might require a hundred million time points, excessive CPU times often occur when the entire system is modelled on the circuit level. The capacity of SystemC to enable system level mixed-signal modelling can vastly reduce simulation times where concepts need to be verified quickly and detailed circuit level modelling is not required. Fig. 8 shows a block diagram of the modelled PLL.

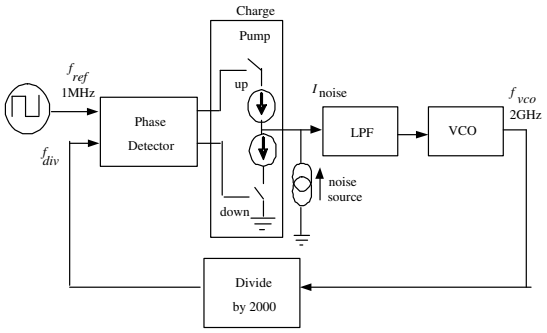

Figure 8: 2GHz Phase Locked Loop with noise and jitter.

5.1 Noise Module

One of the major concerns in the design of PLLs is noise and jitter performance. For example in transceiver designs, jitter from a PLL directly acts to degrade the noise floor and the selectivity of the transceiver. Jitter is modelled here as a Gaussian process with zero mean. It is assumed that only the charge pump was subject to jitter and its signal can be expressed as in Eq. 3,

$$I_{noisy} = I(t + Jitter(t)) \tag{3}$$

PLL noise behaviour is difficult to predict with traditional circuit simulators because a PLL generates repetitive switching events as an essential part of its operation, and the noise performance must be evaluated in time-domain when large signals are present. Most classical simulators, SPICE being the best example, are not capable to simulate noise in PLLs. Using the extended SystemC, suitable noise modules can be constructed with no difficulty. The noise module here relies on the standard C++ random number generator function `rand()` and includes a Box-Muller converter of uniform random numbers to Gaussian distribution.

5.2 Voltage-Controlled Oscillator

The VCO generates a square wave whose frequency is proportional to the input signal level. The VCO frequency is the rate of change of the phase (Eq. 4),

$$\dot{\varphi}(t) = \frac{d\varphi}{dt} = f(v) = f_c + df * V_{filter} \tag{4}$$

Where V_{filter} is the output voltage of the loop filter, f_c is the center frequency of the VCO, and $df = \frac{f_{max} - f_c}{V_{max}}$ is the VCO gain. The VCO was modelled as an equation class and not in circuit level. In the class of the VCO there are methods to add

the VCO contribution to the system jacobian. Equation classes such as VCO class are inherited from the component class so that it allocates its place in the jacobian. Part of the VCO class is shown below:

```
vco::vco(char nameC[5],SystemVariable *node_a, sc_signal<bool>
*Vout): component(nameC,node_a, 0, value){
    Vco=Vout;
    phi = new sc_a_free_variable("phi");
}

void vco::build(void){ ...
    phase = X(phi);
    phase=fmod(phase,1.0);
    if(phase > 1.0)
        PhaseNoisy = phase + Pnoise;
    if (PhaseNoisy > 0.5)
        Vco->write(true);
    if (PhaseNoisy < 0.5)
        Vco->write(false);

    double fmin=0.5e9, fmax=5e9, Vmax=3.3, df, fc=2e9;
    df= (fmax-fc) / Vmax;
    double S, Qdotn, freq;
    S=Sn();
    Qdotn=Xdot(phi);
    freq = fc + df * (a->readn());

    if (freq < fmin || freq> fmax)
    {
        if (freq < fmin )
                freq = fmin;
        else    freq = fmax;

        BuildM(phi,phi,S);
        BuildM(phi,a,0);
        BuildRhs(phi,-Qdotn + fc + (a->readn()) * df);
    }
    else{
        BuildM(phi,phi,S);
        BuildM(phi,a,-df);
        BuildRhs(phi,-Qdotn + fc + (a->readn()) * df);
    }
}
```

5.3 Simulation

The system was simulated with extremely small analogue steps which are required to accurately reflect the effects of noise and jitter. To enforce a step size of 10ps or less, the charge pump module is sensitive to a 100GHz clock, whereas the digital modules are sensitive only to their input signals. Fig. 9 shows different system values in the first micro seconds of the simulation. Table 1 shows some simulation figures.

Table 1: PLL simulation figures

Simulation time	200μ Sec
Number of steps	20 Millions
CPU time	1157.37 Sec

6 CONCLUSION

A mixed-signal simulator based on SystemC has been developed to simulate a general analogue and mixed-signal systems modelled at different abstraction levels. The proposed simulator achieved a good results and capable of operations, some recent simulators are unable to perform. Operations such as noise analysis, reasonable CPU time even with 20 Million time

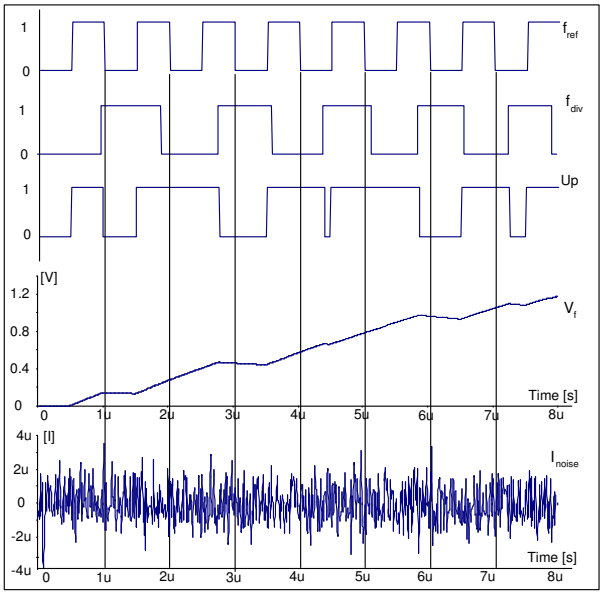

Figure 9: 2GHz Synthesizer simulation results.

steps and modelling at different levels in the same design. The extension is still under further development and is aiming to cover more AMS aspects and more case studies.

References

[1] Pichon F. Blanc S. and Candaele B., "Mixed-signal modelling in vhdl for system-on-chip applications," in *European Design and Test Conference*, Paris, France, 6-9 March 1995, pp. 218–222.

[2] Celoxica, *Survey of System Design Trends*, December 2003.

[3] Open SystemC Initiative OSCI Documents, *SystemC Language Reference Manual*, 2003.

[4] Einwich K. Clauss Ch. Noessing G. Schwarz P. and Zojer H., "Systemc extensions for mixed-signal system design," in *FDL*, Lyon France, 3-7 September 2001.

[5] Bonnerud T. Hernes B. and Ytterdal T., "A mixed-signal functional level simulation framework based on systemc," in *CICC*, San Diego California USA, 6-9 May 2001.

[6] Conti M. Caldari M. Orcioni S. and Biagetti G., "Analog circuit modelling in systemc," in *FDL*, Frankfurt, Germany, 23-26 September 2003.

[7] Grimm Ch. Meise Ch. Heupke W. and Waldschmidt K., "Refinement of mixed-signal systems with systemc," in *DATE*, Messe Munich, Germany, 3-7 March 2003.

[8] Mueller W. Ruf J. Hoffmann D. Gerlach J. Kropf Th. and Rosenstiehl W., "The simulation semantics of systemc," in *DATE*, Messe Munich, Germany, 13-16 March 2001.