SystemC-WMS: A Wave Mixed Signal Simulator

Simone Orcioni, Giorgio Biagetti, and Massimo Conti
Dipartimento di Elettronica, Intelligenza artificiale e Telecomunicazioni
Università Politecnica delle Marche, I-60131 Ancona, Italy
e-mail: SystemC-WMS@deit.univpm.it

Abstract
This paper proposes a methodology for extending SystemC to mixed signal systems, aimed at allowing the reuse of analog models and to the simulation of heterogeneous systems. To this end, a general method for modeling analog modules using wave quantities is suggested, and a new kind of port and channel suitable to let modules communicate via waves have been defined. These entities are plugged directly on top of the standard SystemC kernel, so as to allow a seamless integration with the pre-existing simulation environment, and are designed to permit total interconnection freedom to ease the development of reusable analog libraries.

1 Introduction
SystemC is an emerging tool for the description and simulation of hardware and software at system level [OSCI], and it is not rare that this high level of abstraction could require the interfacing of both digital and analog parts. Such necessity of simulating a continuous-time analog part can arise, for example, in the area of power switching control as in the automotive or RF domains. To this aim it has been proposed [Ein02] to constitute an Open SystemC Initiative (OSCI) Working Group devoted to the development of an extension of SystemC to mixed-signal simulation: SystemC-AMS. In [VGE03a] Vachoux et al. described in detail the SystemC-AMS requirements and objectives. The first aspect considered is the need to encompass a variety of models of computation (MoCs), that can be used in order to describe any kind of system (discrete-event, data-flow, finite-state machines, analog signal flow, generic continuous-time, etc.). Furthermore, SystemC must also extend to heterogeneous domains of application (i.e. electrical, mechanical, fluidic), due to the increasingly complexity of nowadays devices.

The OSCI Working Group claims that SystemC-AMS, besides being suitable for the description and the simulation of heterogeneous systems and supporting continuous-time MoCs, must also meet the following objectives: it must be an extension of the current SystemC; it must provide a (possibly generic) way to handle interactions between MoCs; it must provide appropriate views for the description of continuous-time models; and, finally, it must support the coupling with existing continuous-time simulators. A recent description of the state of the art of this initiative can be found in [VGE03b].

Currently, their implementation is structured into different layers. The solver layer provides simple but efficient solvers for linear differential equations and for explicit-form transfer functions. The synchronization layer provides a simple and fast synchronization scheme that executes analog solvers before the first delta cycle of each time step, scheduling them using static dataflow. Finally, a view layer provides means for specifying equations, for instance using netlists.

In addition to the activity performed by OSCI different papers [BBY03,AK04,BHY01], aimed to the extension of SystemC to analog environments, have been published. In [BBY03] J. Bjørnsen et al. presented a mixed-signal simulation framework oriented to the simulation of signal processing dominated applications. Two pipelined A/D converter architectures and a CMOS Camera-on-a-Chip have been presented as case studies. The library modules proposed do not provide a real continuous-time modeling but
a discrete-time domain regulated by a virtual clock or a multi-phase clocking scheme. The implemented scheduling of analog and mixed-signal processes in this virtual clock environment makes it possible to avoid useless execution of these blocks. More recently, in [AK04] H. Aljunaid and T.J. Kazimierski proposed a mixed-signal extension using a general-purpose analog solver coupled with SystemC kernel via a lock-step synchronization algorithm. This has implied a modification of the SystemC 2.0 kernel to invoke and synchronize an analog solver with it. They used two simulation examples, a boost DC-DC converter and a PLL-based frequency multiplier, to show the effectiveness of the proposed methodology.

The basic SystemC methodology [OSCI] makes use of modules and interfaces to describe complex systems. Modules communicate through interfaces, implemented in a channel, by calling methods in the channel. Conversely, events in the channel can activate modules connected to the channel itself. The present work proposes a methodology for the description of analog blocks using only such instruments and libraries. Taking advantage of this communication scheme and of the underlying SystemC kernel, we implement the various analog parts of a system as analog modules, which communicate by exchanging energy waves through wavechannel interfaces. The use of energy waves permits the definition of a standard analog interface that allows the interconnection of modules belonging to different domains as well as of modules developed independently. Furthermore, interconnections of analog blocks giving rise to simple kinds of nonlinear Differential Algebraic Equations (DAEs) can also be simulated.

2 Description and Modeling of Analog Modules in SystemC

SystemC is essentially a library of C++ classes developed to build, simulate and debug a System on Chip described at system level. It provides an event-driven simulation kernel and the functionality of the system derives from the interaction of concurrent processes, that describe the behavior of individual modules subject to stimuli sent to them by other modules.

The core SystemC simulation paradigm assumes that modules have clearly defined inputs and outputs, and that they communicate between one another by means of appropriate channels. This paradigm allows the simulation to be carried out by a simple time-marching algorithm, that only needs to take care of interactions between modules and the channels directly connected to them, without the need of dealing with the global system topology.

In order to be able to simulate systems containing analog modules some extensions to the base kernel are necessary. In cases where it is easy to obtain a signal flow graph representation of the system, this simulation paradigm can be coupled with an appropriate ODE solver as in [BCCO04], thus enabling an efficient simulation of continuous-time analog modules described by a system of nonlinear ordinary differential equations of the following type:

\[
\begin{align*}
\dot{x} &= f(x, u) \\
y &= g(x, u)
\end{align*}
\] (1)

where \(f\) and \(g\) are vector expressions describing system dynamics, while \(x, y\) and \(u\) are state, output and input vectors, respectively.

Equation (1) should describe a part of the system under consideration, like an \(N\)-port modeled at circuitual level, or it may represent a high level macromodel describing the part functionality. This description is not able to take into account parts that need a DAE system to be described, neither conservative-law systems, however it is quite general, and will thus be used to describe the behavior of a single module.

Nevertheless, a signal-flow-graph representation is not always the most suited to model the interaction between modules representing analog units, since it can be hard to account for load effects or other interactions that might occur as they are interconnected. The goal of the next section is to propose an extension of this approach to cases where a signal-flow-graph representation of the modules is undesirable or excessively demanding.
Figure 1: Example of interconnection problem: (a) two different modules using voltage and current as their input/output signals, (b) some complex adaptor is needed to tie together the two output voltages, (c) electrical port representation using wave quantities, and (d) solution to the interconnection problem by a wave adaptor.

2.1 Module representation with a,b parameters

The first problem that needs to be solved is the possibility of interconnecting modules written independently. Figure 1(a)–(b) depicts one possible problem that can arise trying to bind together electrical modules that use currents or voltages as their input/output signals. Whatever the designer’s choice was regarding what to consider input or output, it would not be possible to simultaneously be able to connect them in series or in parallel, as well as to cascade them.

Furthermore, in non-SFG representations there can be no physical clue on which quantity to consider input or output of a given module. Even if in principle it can be feasible to write a specialized channel that can handle all the possible combinations arising from a random choice, the resulting interconnection model would lack a physical meaning and would likely be cumbersome.

The use of an incident/reflected wave model [Kur65] for the description of analog modules allows us to avoid this difficulty since it can be mandated that modules use incident waves as inputs and produce reflected waves as outputs. This immediately solves the problem for cascaded modules, the parallel or series connection can be accounted for by using an appropriate channel that dispatches waves to the modules it connects together, Fig. 1(d), and permits the formulation of a generic and standard analog interface usable across a variety of domains.

Such channel behaves similarly to the scattering junction of Wave Digital Filters (WDFs) [Fet73], which are digital models of analog filters, obtained through the discretization of individual circuit components via a methodology that can also be extended to circuits in which mildly nonlinear elements are present [SP99].

Our approach uses, like in the WDF theory, the $a,b$ parameters as input/output signals and implements the duties of the scattering junction in a new entity called wavechannel, complying with SystemC conventions for channels. Furthermore, the user can choose the level of abstraction at which to model the system and the integration method (ODE solver) used to solve the continuous time system.

Without loss of generality, we can fix our attention to an $N$-port in the electrical domain, described through its port quantities $v_j$ and $i_j$, $j = 1, \ldots, N$. Figure 1(c) depicts the situation for a single port. The relation between electrical quantities and wave quantities can be obtained from the following definition.
of incident \( (a_j) \) and reflected \( (b_j) \) wave:

\[
a_j = \frac{1}{2} \left( v_j / \sqrt{R_j} + i_j \sqrt{R_j} \right) \\
b_j = \frac{1}{2} \left( v_j / \sqrt{R_j} - i_j \sqrt{R_j} \right)
\]  

(2)

so that \( a_j^2 - b_j^2 \) is the instantaneous power entering port \( j \) and \( R_j \) is a normalization resistance. Similar relations hold for other domains as well. In the frequency domain, this representation leads to the commonly adopted description with a scattering matrix, and the normalization resistance can be assumed like the characteristic impedance of the transmission line connected to the port.

Solving the system (2) for the electrical quantities gives the inversion formulae:

\[
v_j = (a_j + b_j) \sqrt{R_j} \\
i_j = (a_j - b_j) / \sqrt{R_j}
\]  

(3)

that can be useful when translating module descriptions from one set of quantities to the other.

To build in an easy way the representation of an \( N \)-port with the \( a \ b \) parameters we can use the following method. Let us suppose that a port is defined by means of a relation of the type (1), where in the electrical domain \( \{ u, y \} = \{ v, i \} \) (while in other domains we can find, for example, force and velocity or pressure and volume velocity as port variables). Considering the case of a one-port, by using the following relations:

\[
a = \frac{1}{2} (ku + y/k) \\
y = g(x, u)
\]  

(4)

where \( k \) is an appropriate normalization factor, it is possible, by eliminating the variable \( y \), to obtain \( u \) as a function of \( x \) and \( a \):

\[u = h(x, a).
\]  

(5)

Furthermore the reflected wave \( b \) can be written as:

\[b = \frac{1}{2} (ku - y/k).
\]  

(6)

So substituting (5) into (1) and (6) we obtain:

\[
\{ \dot{x} = f_1(x, a) \\
b = g_1(x, a)
\]  

(7)

that are the state space equations written in wave quantities.

### 2.2 Wavechannels

Wavechannels are the means by which modules described by wave quantities communicate. They can be thought of as a bunch of transmission lines connecting ports to a junction box, in which the lines are tied together, and their role is to model the scattering of waves that occurs at the junction.

Consider a junction between \( N \) ports, each with its own normalization factor \( R_j \). Let \( v \) and \( i \) be the voltage and current vector, respectively, and:

\[
\begin{align*}
A_v v &= 0 \\
A_i i &= 0
\end{align*}
\]  

(8)

be a complete and minimal set of Kirchhoff’s equations describing the junction ([\( A_v \)] \( \{ i \}, [A_i] \{ i \} \in \{ 0, \pm 1 \})

We maintain that letting:

\[A_x = A_v \text{ diag } R_k \quad \text{and} \quad A_y = A_i \text{ diag } 1/R_k\]  

(9)
the scattering matrix $S$ (such that $a = S b$), by substituting (3) and (9) into (8), becomes:

$$S = \begin{bmatrix} A_x \times \times A_y \times \times A_y \end{bmatrix}^{-1} \begin{bmatrix} -A_x \\ A_y \\ A_y \end{bmatrix}$$

(10)

where $b$ are the waves reflected by modules and thus entering the junction, whence $a$ are scattered back from the junction to the modules thereby interconnected.

The above formulation can be used for any kind of junction. But, although there are many possible ways in which the lines can be tied together, the most common situation is to have parallel or series connections, as shown in Fig. 2 for channels connecting three ports. From Kirchhoff’s laws, a parallel connection is characterized by the equations:

$$\sum_{j=1}^{N} i_j = 0 \quad v_1 = v_2 = \cdots = v_N$$

(11)

for which the scattering matrix described by (10) results in:

$$a_j = 2 \sum_{k=1}^{N} \frac{b_k}{\sqrt{R_k}} \frac{1}{\sqrt{R_j}} - b_j$$

(12)

Similarly, for a series wavechannel we have:

$$\sum_{j=1}^{N} v_j = 0 \quad i_1 = i_2 = \cdots = i_N$$

(13)

which leads to:

$$a_j = b_j - 2 \sum_{k=1}^{N} \frac{b_k \sqrt{R_k}}{R_j} \frac{1}{\sqrt{R_j}}$$

(14)

It may be worth noticing here that, if $N = 1$, equations (12) and (14) simply become $a_1 = \pm b_1$, and the two channel types are thus able to model the total reflection that takes place at an open circuit or at a shunt, respectively.

In the current implementation of wavechannels the propagation delay can be excluded, so that their connection to instantaneous blocks may result in the production of delay-free loops. This is accounted for by the standard SystemC delta cycle mechanism which, without further intervention, would just use a fixed-point algorithm to search for the solution of the instantaneous loops, provided that the embedded ODE solver does not advance its state while iterating to find the fixed point. The fixed-point solution is equivalent to the solution of Maxwell’s equations in quasi-static conditions, i.e. when it is possible to model the circuit with lumped elements. The quasi-static condition is valid if the wave propagation delay $\tau$ is much smaller than the $\Delta t$ used by ODE solvers. In our fixed-point solution method this $\tau$ is approximated with a null time.

Furthermore, to increase the convergence speed of the fixed-point algorithm, a damping effect has been introduced. This has been done on the basis that, in a time-marching simulation, states between
successive time steps should not be very different, and thus the fixed point solution may take advantage of a limitation in the amount of change allowed to the variables. Let \( a^{(n)}[t] \) be the wave at the \( n \)-th delta cycle of the time step \( t \). The evaluation of the module output functions, based on the values of the inputs \( a^{(n)}[t] \) and of the state \( x[t] \), yields the reflected wave \( b^{(n+1)}[t] \). This is used in equations (12) and (14) to compute the scattering due to interconnections, let us call \( \tilde{a}^{(n+1)}[t] \) the result. We then put:

\[
\tilde{a}^{(n+1)}[t] = a^{(n)}[t] + \lambda(a^{(n+1)}[t] - a^{(n)}[t])
\]  

where \( \lambda \) is a positive constant less than 1 (we obtained good results with values close to 0.9), governing the amount of damping. The update of \( a \) is skipped altogether when the amount of change is below a predefined threshold related to the desired accuracy of the solution, so as to exit from the delta cycling and thus allowing the time to be incremented and the state of ODE solvers to be updated.

With this approach, it has been possible to obtain accurate simulations with a reasonable convergence speed of most of the systems containing delay-free loops provided they do not contain directly coupled state variables, that is, the circuit has a solution for every possible value of the state variables.

2.3 SystemC class library

To ease the implementation of complex systems containing analog blocks, a number of templates and classes have been designed: a new kind of port to let modules communicate via wave quantities (ab_port), a channel that can interconnect them and that does the real computation of the scattering that occurs at junctions (ab_signal), and a template base class (wave_module) that takes care of handling sensitivity lists and port declarations.

Ports expose an interface that allows users to read the incident wave value and to report (write) the reflected wave value, together with utility functions to poll for changes and to get other channel properties:

```cpp
template <class T> struct ab_signal_if : virtual sc_interface
{
    virtual bool poll () const = 0;
    virtual const T read () const = 0;
    virtual void write (const T &t) = 0;
    ...  
};
```

The basic wave_module template looks like the following:

```cpp
template <int n, class T> struct wave_module : sc_module, ...
{
    ab_port <T> port[n];
    sc_event activation;
    ...
};
```

where `port` is the array (or possibly a single variable if \( n=1 \)) of ports used by the module to communicate. Of course, they can freely be mixed with standard SystemC ports. `activation` is an event that is signaled when some change occurs at the waves entering any of the ports, and the template parameter `T` must be associated to a structure, that essentially consists of a collection of typedefs, needed to define the underlying data type used for waves and to document the nature of the port. A number of predefined natures (electrical, mechanical, etc.) have been provided, and, of course, templates to ease the implementation of transducers (that is, modules with ports of different natures) have also been defined and implemented.
Figure 3: Sketch of the execution flow inside modules and wavechannels. The dashed lines are followed by the underlying standard SystemC simulation kernel that schedules the evaluate/update phases in the proper order. Of course, the entire update phase is performed by the wavechannel itself.

A sketch of the flow provided by this library is drafted in Fig. 3. As shown there, with this library the only thing that the user needs to do in order to model an analog module is to implement the state derivative vector field $f$ and output transformation function $g$ as in (7):

```cpp
struct example : wave_module <1, electrical>, analog_module
{
    // state variable x is inherited from analog_module
    void field (double *var) const;
    void calculus ();
    SC_CTOR(example) : analog_module(...) {
        SC_THREAD(calculus);
        sensitive << activation;  
    }
};

void example::calculus ()
{
    x = 0;  // state initialization here
    while (step()) {
        // perform one ODE solver step
        double a = port->read();  // read incident wave here
        double b = g(x, a);       // compute reflected wave
        port->write(b);           // and send it out here
    }
}

void example::field (double *var) const
{
    double a = port->read();
    var[0] = f(x, a);          // evaluate state change
}
```

the step function is inherited from the analog_module and contains a simple time-marching ODE solver [BCCO04]. Currently the user has a choice of Euler and Adams-Bashforth ODE solvers, but the implementation of other time-marching ODE solvers should be straightforward.

Finally, ab_signal takes care of making communication between modules possible. These signals
can just be declared by specifying the nature of the ports and the kind of connection topology to make between them, with an optional default normalization resistance, for instance:

```plaintext
ab_signal <electrical, parallel> test_signal_1(50 ohm);
ab_signal <electrical, series> test_signal_2(10 ohm);
```

and then connected to ports like ordinary SystemC signals.

### 3 Application example

As an example of a possible application of this extension to a real problem a half-bridge inverter has been chosen. A simplified schematic of the circuit is shown in Fig. 4. It is used to drive an RLC load for an induction-heating appliance. The function of the circuit is to regulate the delivery of power to a load. The amount delivered can be set by changing the duty cycle and/or the frequency of the signals controlling the two switches with a proper algorithm that can be implemented in digital hardware. Of course, the maximum output power corresponds to a 50% duty cycle at the resonance frequency, but in the proposed example we have not modeled the details of the digital controller and have thus chosen a 48% duty cycle, for safe operation of the switches, and a fixed frequency of 20 kHz.

The main components of the circuit are the switches (controlled_rectifier), the Graetz’ bridge used to rectify the line voltage (ideal_rectifier), and the voltage source (wave_source) used to
convert stimuli from standard SystemC signals or a signal flow graph representation to the wave representation. Furthermore, there are a couple of different passive reactive linear networks (RC and RLC). All of the analog stuff is connected together by means of wavechannels, as shown in Fig. 5 and by the following code fragment illustrating the structure of the circuit under consideration. A brief description of the most important modules follows the code.

```c
int sc_main (int argc, char *argv[]) {
  sc_signal <electrical::wave_type> in;
  sc_signal <bool> pulse1, pulse2;
  ab_signal <electrical, parallel> mains;
  ab_signal <electrical, parallel> rectified;
  ab_signal <electrical, parallel> chopped;
  ab_signal <electrical, series> shunt;

  generator signal_source("SOURCE1", 230 V, 50 Hz);
  signal_source(in);

  controller ctrl("CONTROL", 20 kHz);
  ctrl(pulse1, pulse2);

  source <electrical> wave_source("SOURCE2", cfg::across);
  wave_source(mains, in);

  ideal_rectifier bridge("BRIDGE");
  bridge(mains, rectified);

  RC filter("FILTER", 1 ohm, 5 uF);
  filter(rectified);

  controlled_rectifier switch1("SWITCH1");
  switch1(chopped, rectified, pulse1);

  controlled_rectifier switch2("SWITCH2");
  switch2(shunt, chopped, pulse2);

  RLC load("LOAD", 3 ohm, 80 uH, 740 nF);
  load(chopped);

  RC snubber("SNUBBER", 10 ohm, 10 nF);
  snubber(chopped);

  sc_start(150e-6, SC_SEC);
  return 0;
}
```

The source class is a generic converter from standard SystemC signals to wave signals, and the second parameter to its constructor specifies an “across”-type (as opposed to a “through”-type) source, that is a voltage source in the electrical domain. The controlled_rectifier module models the behavior of an ideal switch, like a MOS switch with zero on resistance, coupled in parallel with a bypass ideal diode. It has been modeled as a 2-port module with an additional logical input to control the switch. For simplicity, the assumption that normalization resistances are the same for both ports has been made in its formulation, and so imposed in its constructor. That way, in its conducting state, whether it is due to the transistor switched on or to the diode, it simply lets waves through (like a transparent channel), otherwise it reflects them backwards (like a couple of open circuits). Its implementation is shown in the following:
struct controlled_rectifier : wave_module <2, electrical>
{ // Ideal switch with integrated ideal diode
    SC_HAS_PROCESS(controlled_rectifier);
    controlled_rectifier (sc_module_name name);
    void calculus ()
    {
        // sets normalization resistances on both ports to be the same:
        port[0] <<= port[1] <<= 1;
    }
}

void controlled_rectifier::calculus ()
{
    double a0 = port[0]->read(), a1 = port[1]->read();
    bool diode_on = a0 > a1, switch_on = control->read();
    bool on = diode_on || switch_on;
    port[0]->write(on ? a1 : a0);
    port[1]->write(on ? a0 : a1);
}

When the switch is off, the detection of the state of the diode is done by looking at port voltages, but if
the diode is off too, port voltages are proportional to incident waves, since
\[ b_j = a_j \Rightarrow v_j = 2a_j \sqrt{R_0}, \]
so it is perfectly legal to test the latter ones. A similar formulation models the diode bridge in the ideal_rectifier module.

For what concerns the linear components, their are all modeled according to (7) and the example module reported in Section 2.3. In particular the equations governing the RLC circuit in wave quantities are:

\[
\begin{align*}
\dot{x}_0 &= 2a\sqrt{R_0} - (R + R_0)x_0/L - x_1/C \\
\dot{x}_1 &= x_0/L \\
b &= a - x_0\sqrt{R_0}/L 
\end{align*}
\]  
(16)

where the state vector \( x \) is composed of \( x_0 = L \dot{i}_L \) and \( x_1 = C \dot{v}_C \) with obvious meaning of the symbols.

This directly translates into the following state update and output computation functions:

void RLC::field (double *var) const
{
    double a = port->read();
    var[0] = 2*a*sqrt(R0) - state[0]*(R + R0)/L - state[1]/C ;
    var[1] = state[0]/L ;
}

void RLC::calculus ()
{
    R0 = port->get_normalization();
    while (step()) {
        double a = port->read();
        double b = a - state[0]*sqrt(R0)/L;
        port->write(b);
    }
}

that, together with the obvious declaration of the RLC structure, complete the definition of the module. A
model for the RC module can be derived similarly. The complete source code for this example, together with the full library and other applications, will be available from the authors’ web site [BCO].

3.1 Simulation results

The circuit has been simulated using the proposed SystemC extension, which uses a fourth-order Adams-Bashforth ODE solver, and the results compared to a Matlab™ simulation done with the Simulink Power toolbox using the ode15s stiff ODE solver. Excessively long simulation times with the Matlab ode15s solver (Adams-Bashforth), suited for nonstiff systems, led us to believe that testing both simulators with the same solver algorithm could be not very significant because of their different application contexts (which is a single module in our simulator).

Nevertheless, using an adaptive time step with the same maximum $\Delta t$ of 5 ns, we obtained a simulation time of 11.3 s with SystemC-WMS and 2.3 s with Matlab™, both running on an Intel™ Pentium™ M processor at 1000 MHz.

Results are shown in Figs. 6 and 7, where load current and chopped voltage are plotted as a function of time. The curves are completely overlapping.

4 Conclusion

The increasing complexity of systems and circuits ask for an easy way to model and simulate the overall behavior of a complex system spanning multiple domains. In order for SystemC to be able to cope with
these requirements, an extension aimed at allowing the modeling and simulation of analog circuits is mandatory.

This work proposes an effective, and still not excessively complex framework, that simplify the modeling of the interaction between analog models belonging to heterogeneous domains, as well as model reuse. By using power waves as standard input/output signals for analog modules, these can be independently modeled and freely interconnected together in arbitrary topologies without having to deal with complex interface compatibility issues. Moreover, this allows for a uniform treatment of heterogeneous domains, thus paving the way to the development of truly generic and reusable model libraries.

The first results are encouraging in terms of the accuracy of the simulation and, despite the simplicity of the algorithms employed, the variety of the class of circuits that can be simulated.

References


