An Introduction to Modeling Embedded Analog/Mixed-Signal Systems using SystemC AMS Extensions

By
Christoph Grimm, Vienna University of Technology
Martin Barnasconi, NXP Semiconductors
Alain Vachoux, Ecole Polytechnique Fédérale de Lausanne
Karsten Einwich, Fraunhofer IIS/EAS Dresden

June 2008

Abstract

SystemC AMS extensions introduce new language constructs for the design of embedded analog/mixed-signal systems. This paper presents the novel modeling language for analog and mixed-signal functions that supports design and modeling of telecommunications, automotive and imaging sensor applications at various levels of abstraction. A simple example illustrates how these new features facilitate a design refinement methodology for functional modeling, architecture exploration and virtual prototyping of embedded analog and mixed-signal systems.

Introduction

There is a growing trend for tighter interaction between embedded hardware/software (HW/SW) systems and their analog physical environment. This leads to systems in which digital HW/SW is functionally interwoven with analog and mixed-signal blocks such as RF interfaces, power electronics, sensors and actuators, as shown for example by the communication system in Figure 1. We call such systems *Embedded Analog/Mixed-Signal (E-AMS) systems*. Examples of E-AMS systems are cognitive radios, sensor networks or systems for image sensing. A challenge for the development of E-AMS systems is to understand the interaction between HW/SW and the analog and mixed-signal subsystems at architecture level. This requires some means of modeling and simulating the interacting analog/mixed-signal systems and HW/SW systems at functional and architecture levels.

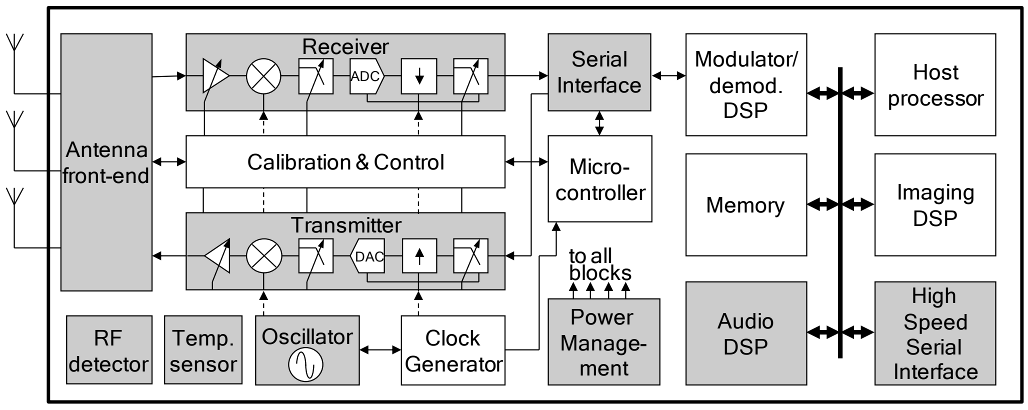


Figure 1: *Example of an embedded analog/mixed-signal architecture: Communication System.*

SystemC [1] supports the refinement of HW/SW systems down to RTL by providing a discrete-event (DE) simulation framework. A methodology for generalized modeling of communication and synchronization that builds on this framework is available: Transaction Level Modeling (TLM) [2] allows designers to perform abstract modeling, simulation and design of HW/SW system architectures. However, the SystemC simulation kernel has not been designed for the modeling and simulation of analog, continuous-time systems and lacks the support of a refinement methodology to describe analog behavior from a functional level down to implementation level.

System-level tools such as Simulink [3] and Ptolemy II [4] are often used for functional modeling and simulation. They may also capture continuous-time behavior, but do not target the design of E-AMS systems at an architecture-level. Hardware description languages (HDLs) such as VHDL-AMS [5] and Verilog-AMS [6] target the design of mixed-signal subsystems close to implementation level, but these languages have limited capabilities to provide efficient HW/SW co-design at high level of abstraction. Existing co-simulation solutions mixing SystemC and Verilog/VHDL-AMS do not provide high enough simulation performances and lack offering a seamless design refinement flow for modeling mixed discrete-event/continuous-time systems and HW/SW systems at architectural level.

In response to these needs from telecommunication, automotive, and semiconductor industries, AMS extensions are introduced for SystemC, providing a uniform and standardized methodology for modeling E-AMS systems. This paper gives an overview of the AMS extensions. We first describe typical use cases and requirements. Then we give an overview of the SystemC AMS language, and how it can be used to create models at different levels of abstraction. We finally illustrate the capabilities with a simple example considering different use cases and design abstractions.

We assume that the reader is familiar with SystemC and has some knowledge of an HDL such as VHDL-AMS or Verilog-AMS and related modeling techniques such as signal flow models, data flow models, and electrical networks.

Use Cases and Requirements

The SystemC AMS extensions are intended to extend the HW/SW oriented SystemC class library to provide a framework for functional modeling, architecture exploration, integration validation, and virtual prototyping of E-AMS systems [7, 8]. Such use cases require a means for modeling and simulation that is more abstract than existing HDLs for analog and mixed-signal systems. At the same time, designers should be able to model AMS components and their interactions with HW/SW systems. The extensions aim to provide a standardized formalism for the modeling of AMS architectures that bridges the gap between functional modeling frameworks and HDLs used for implementation. For this purpose, we consider the following modeling formalisms:

Signal flow models describe AMS systems such as control systems or filter structures at functional and architecture level using block diagrams, assuming continuous time and directed real-valued signals. For simulation, differential and algebraic equations are solved numerically at appropriate time steps. Therefore, simulation of signal flow models is often a time consuming task.

Data flow models are common for modeling DSP algorithms and communication systems at functional and architecture level by processes that communicate via buffers. For simulation, the processes are activated in the data flow's direction, considering different data rates. Note that, in general, data flow models are untimed and can have different semantics. When modeling AMS systems, timed semantics are introduced by assuming discrete time steps between data tokens. Although not as accurate as continuous-time signal flow, discrete-time data flow provides a higher simulation performance with reasonable accuracy.

Electrical networks are – even at functional and architecture level – essential for modeling E-AMS systems. They are required to model loads, protection circuits, and buses at high frequencies using macro models for describing continuous-time relations between voltages and currents. For simulation, differential and algebraic equations are solved, but efficiency can be maintained by using only linear primitives and switches.

Figure 2 gives an overview of modeling formalisms that are required for designing AMS systems and their use cases between functional level and implementation.

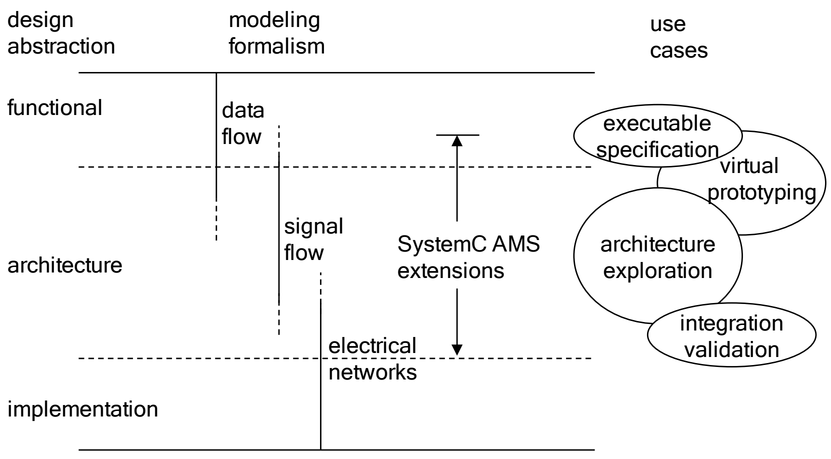


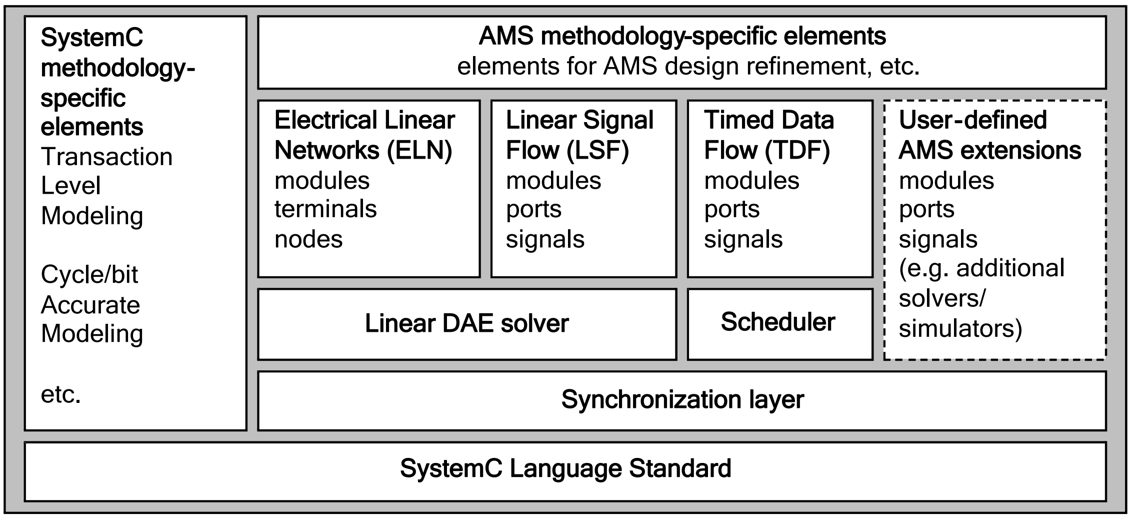
Figure 2: *Modeling formalisms and their use cases between functional level and implementation.*

A major requirement is to maintain an acceptable *simulation performance* while modeling the architecture's behavior with sufficient accuracy. Therefore, the AMS extensions must enable the use of dedicated simulation kernels synchronized with the standard SystemC kernel. Electrical networks require specific simulation capabilities: The simulation of such models needs structural analysis, setup of differential and algebraic equations (DAE), and numerical methods for solving them. Furthermore, the AMS language shall enable the use of dedicated simulation kernels for special cases such as linear networks, which permit a significantly higher simulation performance compared with general nonlinear networks. For data flow models, the special case of synchronous data flow allows the implementation of a dedicated simulation kernel that provides considerable speed-up compared with the SystemC kernel.

Another important requirement is *extensibility*. Industrial design flows for E-AMS systems make use of SPICE-like circuit simulators with high accuracy, special support for RF, nonlinear systems or other application specific extensions [8]. Therefore, the SystemC AMS extensions should allow the industry or EDA vendors to integrate "user-defined extensions", to support different domains such as modeling for telecommunication, automotive, or sensor imaging applications.

Language Structure and Use Cases

The SystemC AMS extensions meet the requirements and use cases discussed in the previous section. They provide support for signal flow, data flow, and electrical networks. The extensions are fully compatible with the SystemC standard as shown in Figure 3. Electrical networks and signal flow models use a linear DAE solver that solves the equation system and which is synchronized with the SystemC kernel. The use of a linear DAE solver restricts networks and signal flow components to linear models in order to provide high simulation performance. Data flow simulation is accelerated using static scheduling that is computed before simulation starts. This schedule is activated in discrete time steps, where synchronization with the SystemC kernel introduces timed semantics. We therefore call it "timed" data flow (TDF).



SystemC methodology-specific elements
Transaction Level Modeling
Cycle/bit Accurate Modeling
etc.
AMS methodology-specific elements
elements for AMS design refinement, etc.
Electrical Linear Networks (ELN)
modules
terminals
nodes
Linear Signal Flow (LSF)
modules
ports
signals
Timed Data Flow (TDF)
modules
ports
signals
User-defined AMS extensions
modules
ports
signals
(e.g. additional solvers/ simulators)
Linear DAE solver
Scheduler
Synchronization layer
SystemC Language Standard

Figure 3: *AMS Extensions for the SystemC Language Standard.*

The SystemC AMS extensions define new language constructs identified by the prefix sca_. They are declared in dedicated namespaces sca_tdf (timed data flow), sca_eln (electrical linear networks), and sca_lsf (linear signal flow) according to the underlying semantics. By using namespaces, similar primitives as in SystemC are defined to denote ports, interfaces, signals, and modules. For example, a timed data flow input port is an object of class sca_tdf::sca_in<type>.

Linear signal flow (LSF) models are specified by instantiating a structure of signal flow primitives such as adders, integrators, differentiators, or transfer functions. These primitives are part of the language definition. The primitives are connected via signals of the class sca_lsf::sca_signal. To access LSF signals from TDF and DE, converter modules have to be instantiated. The instantiation of the primitives can be done in a regular SC_MODULE using the standard SystemC rules. Example 1 gives a simple low pass filter structure with TDF interface, which enables its use in a TDF model.

```
SC_MODULE(lp_filter_lsf)  // Hierarchical models use SC_MODULE class
{
  sca_tdf::sca_in<double>   in;       // Input/outputs in data flow MoC
  sca_tdf::sca_out<double>  out;      //
  sca_lsf::sca_sub*  sub1;            // Subtractor
  sca_lsf::sca_dot*  dot1;            // Differentiator
  sca_lsf::sca_tdf_source*  tdf2lsf1; // TDF -> LSF converter
  sca_lsf::sca_tdf_sink*    lsf2tdf1; // LSF -> TDF converter
  sca_lsf::sca_signal in_lsf, sig, out_lsf; // LSF signals
  lp_filter_lsf(sc_module_name, double fc=1.0e3) // Constructor with parameters
  {
    tdf2lsf1 = new sca_lsf::sca_tdf_source("tdf2lsf1"); // TDF->LSF converter
      tdf2lsf1->inp(in);              // Port/signal binding like SystemC
      tdf2lsf1->y(in_lsf);
    sub1 = new sca_lsf::sca_sub("sub1");
      sub1->x1(in_lsf);
      sub1->x2(sig);
      sub1->y(out_lsf);
    dot1 = new sca_lsf::sca_dot("dot1",1.0/(2.0*M_PI*fc)); // M_PI=3.1415…
      dot1->x(out_lsf);
      dot1->y(sig);
    lsf2tdf1 = new sca_lsf::sca_tdf_sink("lsf2tdf1");   // LSF->TDF converter
      lsf2tdf1->x(out_lsf);
      lsf2tdf1->outp(out);
  }
};
```

Example 1: *LSF model of a low pass filter structure with TDF converter modules.*

Timed data flow (TDF) models consist of TDF modules that are connected via TDF signals using TDF ports. Connected TDF modules form a contiguous structure called TDF cluster. Clusters must not have cycles without delays, and each TDF signal must have one source. A cluster is activated in discrete time steps. The behavior of a TDF module is specified by overloading the predefined methods `set_attributes()`, `initialize()`, and `processing()`:

- The method `set_attributes()` is used to specify attributes such as rates, delays or time steps of TDF ports and modules.
- The method `initialize()` is used to specify initial conditions. It is executed once when the simulation starts.
- The method `processing()` describes time-domain behavior of the module. It is executed at each activation of the TDF module.

It is expected that there is at least one definition of the time step value and, in the case of cycles, one definition of a delay value per cycle. TDF ports are single-rate by default. It is the task of the elaboration phase to compute and propagate consistent values for the time steps to all TDF ports and modules. Before simulation, the scheduler determines a schedule that defines the order of activation of the TDF modules, taking into account the rates, delays, and time steps. During simulation, the `processing()` methods are executed at discrete time steps. Example 2 shows the TDF model of a mixer. The `processing()` method will be executed with a time step of 1µs.

```
SCA_TDF_MODULE(mixer) // TDF primitive module definition
{
  sca_tdf::sca_in<double>  rf_in, lo_in; // TDF in ports
  sca_tdf::sca_out<double> if_out;       // TDF out ports
  void set_attributes()
  {
    set_timestep(1.0, SC_US);            // time between activations
  }
  void processing()                      // executed at each activation
  {
    if_out.write( rf_in.read() * lo_in.read() );
  }
  SCA_CTOR(mixer) {}
};
```

Example 2: *TDF model of a mixer.*

In addition to the pure algorithmic or procedural description of the `processing()` method, different kind of transfer functions can be embedded in TDF modules. Example 3 gives the TDF model of a gain controlled low pass filter by instantiating a class that computes a continuous-time Laplace transfer function (LTF). The coefficients are stored in a vector of the class `sca_util::sca_vector` and are set in the `initialize()` method. The transfer function is computed in the `processing()` method by the `ltf` object at discrete time points using fixed-size time steps.

```
SCA_TDF_MODULE(lp_filter_tdf)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;
  sca_tdf::sc_in<double>   gain;         // converter port for SystemC input
  sca_tdf::sca_ltf_nd ltf;               // computes transfer function
  sca_util::sca_vector<double> num, den; // coefficients
```

```
    void initialize()
    {
      num(0) = 1.0;
      den(0) = 1.0;
      den(1) = 1.0/(2.0*M_PI*1.0e4); // M_PI=3.1415…
    }
    void processing()
    {
      out.write( ltf(num, den, in.read() ) * gain.read() );
    }
    SCA_CTOR(lp_filter_tdf) {}
  };
```

Example 3: *Continuous-time Laplace transfer function in a TDF module.*

The TDF modules given in Examples 2 and 3 can be instantiated and connected to form a hierarchical structure together with other SystemC modules. The TDF modules are connected as in SystemC with TDF signals (`sca_tdf::sca_signal<type>`) and SystemC signals as shown in Example 4.

```
SC_MODULE(frontend)  // SC_MODULES used for hierarchical structure
{
  sca_tdf::sca_in<double>     rf, loc_osc; // use TDF ports to connect with
  sca_tdf::sca_out<double>    if_out;      // TDF ports/signals in hierarchy
  sc_core::sc_in<sc_dt::sc_bv<3> > ctrl_config; // SystemC input agc_ctrl configur.
  sca_tdf::sca_signal<double> if_sig;      // TDF internal signal
  sc_core::sc_signal<double>  ctrl_gain;   // SystemC internal signal
  mixer*         mixer1;
  lp_filter_tdf* lpf1;
  agc_ctrl*      ctrl1;
  SC_CTOR(frontend)
  {
    mixer1 = new mixer("mixer1");
      mixer1->rf_in(rf);
      mixer1->lo_in(loc_osc);
      mixer1->if_out(if_sig);
    lpf1 = new lp_filter_tdf("lpf1");
      lpf1->in(if_sig);
      lpf1->out(if_out);
    ctrl1 = new agc_ctrl("ctrl1");  // SystemC module
      ctrl1->out(ctrl_gain);
      ctrl1->config(ctrl_config);
  }
};
```

Example 4: *Structural description, including TDF and DE modules.*

Predefined converter ports (`sca_tdf::sc_out` or `sca_tdf::sc_in`) can establish a connection to a SystemC DE channel, e.g. `sc_signal<T>`, reading or writing values during the first delta cycle of the current SystemC time step. Example 5 illustrates the use of such a converter port in a TDF module modeling a simple A/D converter with an output port to which a SystemC DE channel can be bound.

```
SCA_TDF_MODULE(ad_converter)                   // Very simple AD converter
{
  sca_tdf::sca_in<double>  in_tdf;             // TDF port
  sca_tdf::sc_out<sc_dt::sc_int<12> > out_de;  // converter port to DE domain
  void processing()
  {
    out_de.write( static_cast<sc_dt::sc_int<12> >(in_tdf.read() ) );
  }
  SCA_CTOR(ad_converter) { }
};
```

Example 5: *TDF model of a simple A/D converter.*

Electrical linear networks (ELN) are specified by instantiating predefined network primitives such as resistors or capacitors. As with LSF, the primitives can be instantiated in a regular SC_MODULE. To access the voltages or currents in the network, converters have to be used. The SystemC AMS extensions provide converters that translate these voltages and currents to double-precision floating point values in timed data flow and discrete-event models and vice versa. Example 6 gives the ELN model of a low-pass filter implemented with one resistor primitive and one capacitor primitive. Since it is intended to use the model in a TDF context, additional converter primitives are used for converting TDF data sample values to voltages and vice-versa:

```
SC_MODULE(lp_filter_eln)
{
  sca_tdf::sca_in<double>  in;
  sca_tdf::sca_out<double> out;
  sca_eln::sca_node  in_node, out_node;   // node declarations
  sca_eln::sca_node_ref  gnd;            // reference node
  sca_eln::sca_r  *r1;                   // resistor
  sca_eln::sca_c  *c1;                   // capacitor
  sca_eln::sca_tdf_vsource *v_in;        // converter TDF -> voltage
  sca_eln::sca_tdf_vsink   *v_out;       // converter voltage -> TDF
  SC_CTOR(lp_filter_eln)
  {
    v_in = new sca_eln::sca_tdf_vsource(“v_in”, 1.0); // scale factor 1.0
      v_in->inp(in);                     // TDF input
      v_in->p(in_node);                  // is converted to voltage
      v_in->n(gnd);
    r1 = new sca_eln::sca_r(“r1”, 10e3);  // 10kOhm resistor
      r1->p(in_node);
      r1->n(out_node);
    c1 = new sca_eln::sca_c(“c1”, 100e-6);// 100uF capacitor
      c1->p(out_node);
      c1->n(gnd);
    v_out = new sca_eln::sca_tdf_vsink(“v_out”, 1.0); // scale factor 1.0
      v_out->p(out_node);                // filter output as voltage
      v_out->n(gnd);
      v_out->outp(out);           // here converted to TDF signal
  }
};
```

Example 6: *Continuous-time transfer function in a TDF module.*

In addition to a means for modeling linear signal flow, timed data flow, and linear electrical networks the SystemC AMS extensions provide a means for tracing signals.

A Modeling and Methodology Example

To demonstrate the capabilities of the AMS extensions, a simplified yet illustrative example is presented. The use of the AMS extensions is explained for the use cases shown in Figure 2:

- executable specification at functional level,
- architecture exploration at functional level, architecture level, and implementation level,
- integration validation, and
- virtual prototyping.

An *executable specification* is made to verify the correct understanding of the system specification by simulation. For this use case, data flow and signal flow are appropriate modeling styles. Note, that the executable specification introduces structures and algorithms that are hard to subsequently change or modify later. Therefore, structures and algorithms that match the architecture should be chosen carefully.

The specification of the simple example is shown in Figure 4. It is the concept of a very basic binary amplitude shift keying (BASK) receiver that consists of a mixer, a low pass filter, and a BASK demodulator.

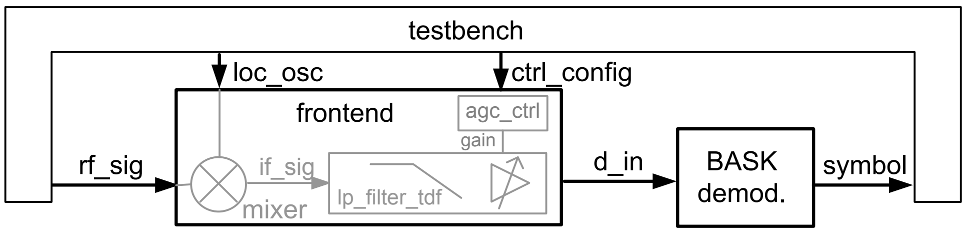


Figure 4: *BASK receiver at functional level.*

To verify the BASK receiver specification, it is modeled using TDF. To achieve good simulation performance we use TDF for the whole BASK receiver model. We obtain data samples by oversampling the assumed continuous-time signals and represent them by double-precision values in the frontend. We also assume that the BASK demodulator generates one symbol for every 20000 input samples. Note that we use TDF signals to model continuous-time signals in the frontend. Example 7 gives the TDF model of the BASK demodulator, assuming a DSP-based implementation.

```
SCA_TDF_MODULE(bask_demodulator)
{
  sca_tdf::sca_in<double> in;     // input samples
  sca_tdf::sca_out<bool>  out;    // output symbol (bit)
  void set_attributes()
  {
    in.set_rate(20000);           // port in has 20000 samples/timestep
    out.set_timestep(0.1, SC_MS);
  }
  void processing()               // maps 20000 samples to 1 symbol
  {
    double val = 0.0;
    for (unsigned long i = 0; i < in.get_rate(); i++)
      val += abs( in.read(i) );
    if ( val>THRESHOLD ) out.write( true ); // THRESHOLD defined in header
    else out.write( false );
  }
  SCA_CTOR(bask_demodulator) {}
};
```

Example 7: *Executable specification of BASK demodulator.*

Using the frontend module given in Example 4, the TDF module of the BASK demodulator given in Example 7 and another module for the testbench not given here, the structure of a model that can be used to evaluate the functional specification in Figure 3, is given in Example 8:

```
#include “systemc-ams”         // top level
#include “bask-demodulator.h”  // other definitions
int sc_main(int argc, char* argv[]) {
  sca_tdf::sca_signal<double> rf, loc_osc, d_in;
  sca_tdf::sca_signal<bool>   symbol;
  sc_core::sc_signal<sc_dt::sc_bv<3> > ctrl_config;
  frontend fe1(“fe1”);
    fe1.loc_osc(loc_osc); fe1.ctrl_config(ctrl_config);
    fe1.rf(rf); fe1.if_out(d_in);
  bask_demodulator bask1(“bask1”);
    bask1.in(d_in);
    bask1.out(symbol);
  testbench tb1(“tb1”);
    tb1.rf(rf); tb1.loc_osc(loc_osc); tb1.ctrl(ctrl_config);
    tb1.symbol(symbol);
  // tracing, ...
  sc_core::sc_start(51.2, SC_MS);
};
```

Example 8: *Executable specification of BASK receiver shown in Figure 3 (top level).*

The *architecture exploration* stage evaluates and determines the key properties of the architecture. It is structured in two phases: In the first phase, the executable specification is refined by adding the non-ideal properties of an implementation to get a better understanding of their impact on the overall system behavior. In the second phase, the architecture's structure and interfaces are adapted to match the final implementation to get a more accurate model. Note, that this also implies the use of different models of computation. For example, HW/SW architectures can be analyzed more accurately using cycle accurate or TLM modeling.

In this example, the impact of noise and distortions of the mixer is evaluated in the first phase. The `processing()` method of the mixer in the executable specification is refined by adding a simple behavioral model of noise (function `my_noise()`) and by C++-code that models third-order non-linear distortion as illustrated in Example 9.

```
void processing()  // Mixer refined with distortions and noise
{
  double rf = rf_in.read();
  double lo = lo_in.read();
  double rf_dist  = ( alpha – gamma * rf * rf ) * rf; //alpha and gamma
  double mix_dist = rf_dist * lo;                     //user-defined
  if_out.write( mix_dist + my_noise() );
}
```

Example 9: *Refinement of the ideal mixer (Example 2) by adding non-linear distortion and noise.*

Other parameters such as bit widths, delays, time steps and rates can also be easily analyzed by refinement of the executable specification. For evaluation of different sample rates, delays and time steps, the properties specified in the attributes section of TDF modules can be modified. The impact of bit widths and quantization errors can be analyzed by replacing the floating point data types in the `processing()` methods with `sc_int<bw>` finite-length integer types. In Example 10, different bit widths are evaluated by replacing the double-precision data type with a 16 bit integer representation.

```
void processing()  // Demodulator refined with quantization
{
  sc_dt::sc_int<16> val;  // evaluate 16 bit accuracy
  for (unsigned long i = 0; i < in.get_rate(); i++)
    val += abs(static_cast<sc_dt::sc_int<16> > ( in.read(i) ) );
  if ( val > THRESHOLD ) out.write(true); // THRESHOLD defined in header
  else out.write(false);
}
```

Example 10: *Refinement of simple BASK demodulator to evaluate quantization.*

In the second phase of architecture exploration the intended architecture is modeled more accurately: E.g. the low pass filter could be specified by an electrical network (Example 6 or a more complex one), and the simple DSP function with a better and more complex one. With knowledge of the required sample rates and bit widths, an A/D converter between the low pass filter and the BASK demodulator is added. This determines the A/D partitioning as shown in Figure 5 and can be seen as a starting point for HW/SW co-design using SystemC. Instead of a full TDF model, a model that uses an A/D converter to translate the TDF signal `d_in` to a cycle accurate or TLM signal `d_in_de` can be developed. The interface to and modeling of the demodulator HW/SW system using pure SystemC is not covered in this white paper.

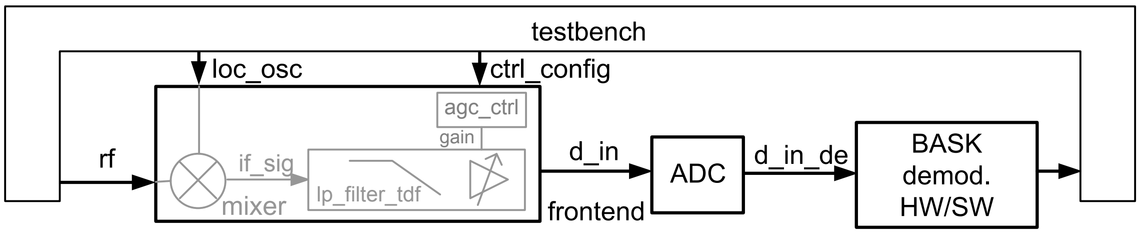


Figure 5: *Architecture level model of BASK receiver with ADC and a BASK HW/SW system.*

To prepare the design of subsystems and *integration validation*, the interfaces of all subsystems must be modeled accurately: The interfaces and data types used in the models should match the implementation. For analog circuits this relates to electrical nodes. For digital circuits, this relates to pin accurate buses. For HW/SW systems TLM interfaces might be appropriate. After the definition and design of the analog, digital HW and SW components, these components are integrated and their correctness is verified within the overall system.

The layered architecture of the SystemC AMS extensions encourages its use as a simulation backplane in which other simulators can be integrated via their C-language interfaces. In the BASK receiver example, the low pass filter and the mixer could be replaced by circuit level models to validate connectivity and integrity of the subsystems. The AMS extensions are not intended to replace circuit simulators. The use of more accurate circuit simulators to verify circuit implementations can be applied, once an integration of a circuit simulator is available.

Virtual prototyping provides software developers with a high-level untimed or timed model that represents the hardware architecture. Especially for E-AMS systems, where software is interacting with AMS hardware, interoperability with SystemC TLM extensions is important. In the BASK receiver example, the refined TDF model combines high simulation speed with appropriate accuracy to act as a virtual prototype for further development of the HW/SW subsystem. Then, the AMS subsystem becomes part of the virtual prototype.

For systems that are more complex and realistic than the simple BASK example, SystemC AMS extensions support a similar refinement methodology to SystemC that can be applied to the overall system by introducing hierarchical converter channels or adapters [9].

Summary

The SystemC AMS extensions enhance the available SystemC standard with support for linear electrical networks, linear signal flow, and timed data flow models to efficiently model and simulate analog mixed-signal architectures. New language constructs support the creation of analog and mixed-signal models at higher levels of abstraction, introducing functional and architecture-level modeling. This enables a design refinement methodology for executable specification, architecture exploration, integration validation, and virtual prototyping of E-AMS systems in a seamless, C-based environment. Its extensibility supports application specific requirements, e.g. to model nonlinear or radio frequency behavior.

This white paper has intentionally omitted other aspects of the SystemC AMS extensions to keep it concise. Most notably, the support for small signal analysis and a means for tracing have not been addressed. These features are described in the Language Reference Manual of the SystemC AMS extensions.

Extending the unique capabilities of SystemC with new AMS features offers a powerful modeling and simulation framework, enabling design and verification for a wide range of applications and systems.

References

[1] IEEE Std. 1666-2005. IEEE Press.

[2] Frank Ghenassia (editor): *Transaction Level Modeling with SystemC*, Springer 2005.

[3] The Mathworks Simulink; http://www.mathworks.com/products/simulink

[4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, *Taming Heterogeneity---the Ptolemy Approach*, Proceedings of the IEEE, v.91, No. 2, January 2003.

[5] IEEE Std. 1076.1-2007. IEEE Press.

[6] Accellera: *Verilog-AMS Language Reference Manual* Version 2.2, 2004. Analog & Mixed-Signal Extensions to Verilog HDL; http://www.verilog.org/verilog-ams/

[7] OSCI AMS Working Group, *SystemC AMS extensions Requirements Specification,* 2007.

[8] Alain Vachoux, Christoph Grimm, Karsten Einwich: *SystemC Extensions for Heterogeneous and Mixed Discrete/Continuous Systems.* In: International Symposium on Circuits and Systems 2005 (ISCAS '05), Kobe, Japan. IEEE, May 2005.

[9] Christoph Grimm: *Modeling and Refinement of Mixed Signal Systems with SystemC*. In: SystemC: Methodologies and Applications. Kluwer Academic Publisher (KAP), June 2003.