Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems

Enrico Fraccaroli, Michele Lora, Sara Vinco*, Davide Quaglia and Franco Fummi
Department of Computer Science, University of Verona, Italy, Email: name.surname@univr.it
*Department of Computer Engineering, Politecnico di Torino, Italy, Email: sara.vinco@polito.it

***Abstract*—Nowadays, the design of applications based on smart systems requires the joint simulation of both digital and analog aspects. Even if analog-mixed-signal (AMS) extensions of hardware description languages are an enabling factor, they do not provide a general methodology for the integration of AMS models into digital virtual platforms. This paper defines the problem and provides two main contributions: *1)* the automatic conversion of analog models from Verilog-AMS to C++/SystemC, to remove the overhead of co-simulation with traditional virtual platform tools, and *2)* the automatic abstraction of analog conservative models, with the goal of increasing simulation speed. Experimental results show that the virtual platform with automatically integrated analog components is 40 times faster than co-simulation with Verilog-AMS, and the increase of speed due to abstraction is more than 100%.**

***Index Terms*—Heterogeneity, analog components, signal flow**

I. Introduction

Smart systems are small intelligent devices in which the capability to interact with the physical environment is pushed to the extreme, thus opening unprecedented opportunities in home/building/factory automation, automotive, wearable and healthcare devices, as well as lab-on-chip. Figure 1 shows the general architecture of a smart system. In such systems, digital hardware components executing software are tightly coupled with analog components like sensors (e.g., accelerometers) and actuators (e.g., micro-mirrors or micro-fluidic devices).

As software applications have become the primary product differentiator, systems companies now expect their EDA suppliers to provide not only silicon, but also complete *virtual platforms* for developing software on top of a simulated version of actual hardware [9]. The creation of a virtual platform thus consists in providing simulation models for each component of the smart system, to allow accurate estimation of system behavior. The main issue in the creation of such platforms is minimizing simulation time to make software execution as fast as on the actual system.

As the main objective of virtual platforms is software development rather than hardware design, simulation efficiency is obtained by rising the abstraction level of the hardware description. This process is well-established and entirely automated for digital hardware: RTL descriptions are abstracted to TLM, data types avoid multi-value logic, and software execution is modeled at instruction level, thus sacrificing cycle accuracy [3].

Unfortunately, the presence of analog components introduces new challenges for virtual platform creation. Tools for the design of analog components typically generate descriptions based on differential and algebraic equations, thus requiring equation resolution instead of event-driven simulation. This makes *simulation time* a critical dimension. Even worse, environments for the representation of analog hardware (e.g., Verilog-AMS [11], SystemC-AMS, Matlab, and SPICE [6]) enable co-simulation of digital and analog hardware, at the price of a further increase of simulation time (see Table VI in [13]).

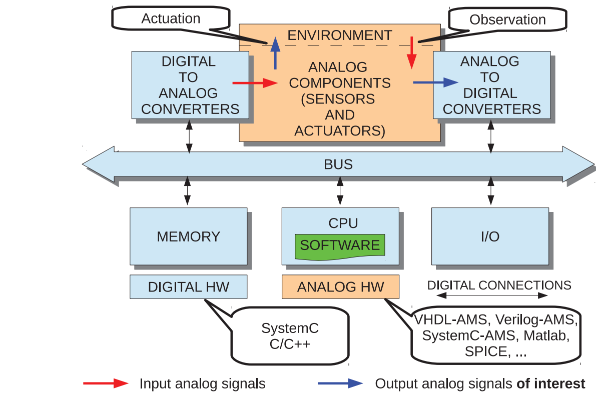

Fig. 1. General architecture of a smart system.

It is thus necessary to build *an abstraction methodology for analog models*, to reduce simulation time and simplify the construction of virtual platforms. As represented by red/blue arrows in Figure 1, the analog subsystem is stimulated by analog input signals while only a subset of its output signals is either observed by the digital HW or used to affect external environment. The paper exploits this fact, by proposing an automatic methodology for extracting the input/output relationships of interest, to generate abstracted signal flow representations suitable for homogeneous simulation with digital components. Therefore, simulation speed is increased by both reducing model complexity and removing co-simulation.

The paper is organized as follows. Background concepts and related work are reported in Section II. The problem is formulated in Section III, and the abstraction methodology is described in Section IV. Experimental results are reported in Section V, and Section VI draws our conclusions.

II. Background and State of the Art

A. AMS extensions of hardware description languages

Verilog-AMS and VHDL-AMS extend traditional HDL languages to analog and mixed signal systems. Despite of

978-3-9815370-6-2/DATE16/ ©2016 EDAA

the syntactic differences, both languages represent the same systems and constructs [11]. In the following Sections, we will use Verilog-AMS syntax, however all considerations are applicable to VHDL-AMS.

Verilog-AMS models a system as a topology of nodes, associated with potential (*i.e.*, voltage, `V()`) and flow quantities (*i.e.*, current, `I(branch)`). The behavior of the system is thus described in terms of relationships between nodes through algebraic and differential equations, called *contribution statements* (denoted with `<+`).

Verilog-AMS supports both *conservative* and *signal-flow* descriptions by using the concepts of nets, nodes, branches, and ports. Conservative descriptions are made of explicit equations (*i.e.*, contribution statements modeled by the designer) and by implicit equations, *i.e.*, energy conservation laws. The simulation of Verilog-AMS models often relies on SPICE (or derived simulators [10]). This makes AMS simulation very accurate but slow, thus not allowing an effective simulation of more complex systems, *e.g.*, including embedded software running on a processor core [1].

SystemC-AMS extends SystemC (also denoted as SystemC-DE for its discrete-event model of computation) to represent analog and mixed-signal systems. It has been successfully adopted in many simulation scenarios [12]. SystemC-AMS provides three different abstraction levels, supporting different communication styles and different adherence to the physical domain. *Electrical Linear Network* (ELN) models electrical networks through the instantiation of predefined primitives, *e.g.*., resistors or capacitors, associated with electrical equations. *Linear Signal Flow* (LSF) adopts signal-flow (i.e., non conservative) representations but still supports differential equation. The SystemC-AMS internal solver analyses the ELN and LSF components to derive the equations describing system behavior, that are solved to determine system state at any simulation time. *Timed Data-Flow* (TDF) models are signal-flow representations scheduled statically by considering their producer-consumer dependencies.

B. Virtual platforms for effective smart system design

Virtual platforms are a powerful solution in embedded system design, as they allow to explore alternative design solutions, to achieve early validation of the overall system and to develop applications on top of HW components. Most of the currently available virtual platform environments target simulation speedup by adopting C++ and SystemC [9]. On one hand, C++ native constructs and types allow a lightweight and fast execution. On the other, SystemC, particularly in its TLM extension, eases integration and reuse of existing IPs, still preserving good simulation performance and adherence with respect to the IP functionality. Note that the choice of a single language supporting the overall simulation is crucial, as this avoids the overhead induced by co-simulation, as a result of frequent synchronization between different simulators [7].

C. Techniques for model conversion and abstraction

State of the art techniques for conversion and abstraction of analog models do not remove the energy conservative property which, if not needed, wastes simulation speed. The approach in [13] avoids co-simulation by proposing language conversion from Verilog-AMS to SystemC-AMS/ELN. The approach in [8] approximates non-linear analog circuit models as a set of linear models. Model Order Reduction [2] reduces the state space size of large-scale dynamical models. No sound methodology exists to translate AMS components into C++ or SystemC-DE. This reduces the efficiency of virtual platforms.

```
`include "constants.vams"
`include "disciplines.vams"
module ActiveFilterOpAmp (in, out);
     input  in;
     output out;
     electrical inp, in, out, gnd, n1, n2;
     parameter real R1 = 400;
     parameter real R2 = 1.6e03;
     parameter real Ri = 1e09;
     parameter real Ro = 125;
     parameter real C1 = 2e-07;
     parameter real Gain  = 1e09;
     parameter real lockrange = 3; // lockrange according input voltage
     real vin;
     analog begin
          V(inp)  <+ 12 * sin(100 * `M_TWO_PI * $abstime);
          V(gnd) <+ 0.0;
          vin = V(inp);
          if (vin > lockrange)
               vin = lockrange;
          else if (vin < -lockrange)
               vin = -lockrange;
          V(in)  <+ vin;

          I(in , n1)  <+ V(in , n1)  / R1;
          I(n1 , out) <+ ddt(V(n1,out)) * C1;
          I(n1 , out) <+ V(n1 , out) / R2;
          I(n1 , gnd) <+ V(n1 , gnd) / Ri;
          V(n2 , gnd) <+ V(n1 , gnd) * Gain;
          I(out, n2)  <+ V(out, n2)  / Ro;
     end
endmodule
```

Fig. 2. Verilog-AMS description of an active filter (Figure 8).

III. Problem Statement

This work addresses the problem of generating efficient C++/SystemC code from Verilog-AMS descriptions for its integration into virtual platforms without the need of co-simulation. As confirmed by analyzing a set of Verilog-AMS models [1], analog descriptions may consist of different kinds of blocks, i.e., declarations (e.g., block *a* in Figure 2), signal-flow representations (e.g., block *b*), and conservative representations (e.g., block *c*).

In the following, let U be the set of input stimuli, Y the set of output signals, X the set of internal state variables of the analog sub-system, and X_0 their value at the beginning of simulation.

A. Signal flow representation

Components with signal flow representation are modeled with equations that describe output signals of interest as a function of both input signals and the internal state of the system:

$$\begin{cases} X(t) &= f_0(X_0) + f_1(0, t, U(t)) \\ Y(t) &= g(X(t), U(t)) \end{cases} \quad (1)$$

where functions $f_0()$, $f_1()$, and $g()$ can be built in Verilog-AMS by using numerical/logical operators, conditional statements (e.g., if-else), math functions (e.g., exp(x), sin(x)), and analog operators (e.g., derivative, integral).

[1] For example http://www.designers-guide.org/VerilogAMS/ and http://www.eda.org/verilog-ams/htmlpages/examples.html

B. Conservative representation

Conservative representation consists of a *graph with a set on nodes connected by branches*. Such graph can be either an electrical network (e.g., resistors, capacitors, inductances, and generators connected by wires) or a bond graph [4] for multi-domain physical systems. For each branch a potential and a flow are defined (e.g., voltage and current in the electric domain) so that their product has power dimension. For each branch, the so-called *dipole equation* gives the relationship between potential and flow quantities (i.e., voltage and current in the electric domain). The Verilog-AMS simulator adds energy conservation laws to the dipole equations depending on graph topology, thus obtaining a system of differential and algebraic equations:

$$\begin{cases} \frac{d}{dt}g(X(t)) & = & f(X(t)) + h(U(t)) \\ Y(t) & = & k(X(t)) + q(U(t)) \end{cases} \quad (2)$$

where functions $f()$, $g()$, $h()$, $k()$, and $q()$ can be built in Verilog-AMS by using numerical/logical operators and math functions (e.g., exp(x), sin(x)).
The simulation of such models requires to solve Equation 2 thousands of times per second to capture transient behavior. Unfortunately, the sparse linear solver and device evaluation are two most serious bottlenecks in this kind of simulators [5]. This has a major impact on conservative representations, as all branches must be computed to consider energy conservation at any time, while signal flow representations only compute output signals of interest.

C. Abstraction and conversion of Verilog-AMS models

The problem of generating C++/SystemC code should be addressed in a different way for signal flow and conservative representations.
In case of signal flow description, the conversion problem consists in finding a C++/SystemC counterpart of the syntax elements contained in Equation 1 and writing the translated equations in the same order as their original counterparts appear in the Verilog-AMS description.
Vice versa, conservative descriptions contain all the physical aspects of the system, so that energy conservation can be represented. Nevertheless, with reference to the architecture represented in Figure 1, only a subset of analog output signals affects the behavior of digital hardware and software. Several computations can thus be avoided by restricting the evaluation only to output signals of interest. It is important to note that the loss of information about energy conservation is not relevant, as virtual platforms do not aim at supporting design and verification of analog components.
Figure 3 exemplifies this concept on the solution of the set of input-state-output equations in Equation 2, implemented as a signal flow description. Bold arrows and gray boxes represent the sub-set determining the evaluation of the sole output y_1. The extraction of this sub-set is considered as model abstraction since the resulting representation contains less information but requires less computational effort to be simulated. Information loss can be controlled during the abstraction process, by deciding the output signals of interest.

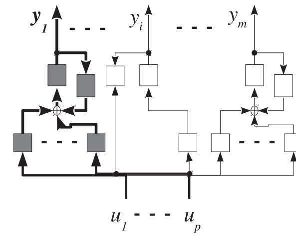

Fig. 3. Mapping of the complete set of input-state-output equations of a conservative model onto a block diagram and, in gray, the sub-set related to output y_1.

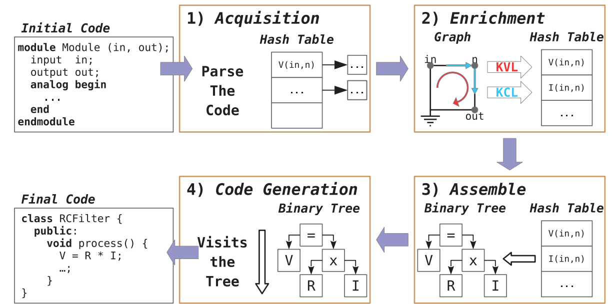

Fig. 4. Complete abstraction flow.

The extracted signal flow representation can be easily converted into SystemC-AMS/TDF, SystemC-DE, and pure C++ to be seamlessly integrated into C++/SystemC virtual platforms, thus avoiding co-simulation with analog solvers.
It is worth noting that the proposed methodology can be also applied to analog models derived from piecewise linear models [8] so that this kind of non-linearity can be easily covered. Furthermore, with respect to Model Order Reduction, the proposed methodology preserves the original state space thus guaranteeing a higher or equal accuracy.

IV. Abstraction Methodology

This section explains the details of the abstraction methodology. Since it refers to an electrical linear network, the description of the algorithm will use the corresponding terminology. However the approach can be also applied to other physical domains if they can be mapped onto electrical linear networks.

The proposed abstraction methodology is outlined in Figure 4. The parameters of the algorithm are the output signal of interest and the description of the electrical linear network as an arbitrary set of constitutive dipole equations.

A. Step 1 - Acquisition

Dipole equations are stored in an optimized data structure, i.e., a Multimap, with average-case insertion time $O(1)$ and with search and deletion time proportional to list lenghth $O(l)$.

The right side of each dipole equation, containing values, variables, operators and functions, is parsed into an abstract syntax tree (AST) representation (values and variables are leaves of the tree whereas operators are intermediate nodes). Every element of the tree has a set of associated flags for storing additional information, e.g., the presence of a derivative or integrative operator. From the same set of dipole equations, the acquisition tool retrieves information concerning the topology

Program **Input** : Hash table containing the dipole equations and the circuit's graph.
Output: An enriched version of the original hash table.

```
2  ENRICHMENT(HT, G)  NodalAnalysis(HT, G);
3  MeshAnalysis(HT, G);
4  foreach equation in HT do
5      Equation previous = equation;
6      foreach term in equation.terms do
7          Equation solved = Solve(equation, term);
8          HT.insert(solved);
9          previous.nextDependent = solved;
10         previous = solved;
11     previous.nextDependent = equation;
12 return HT
```

Algorithm 1: Enrichment of the set of equations.

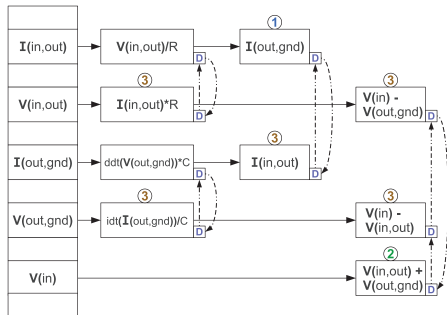

Fig. 5. Final hash table, with reference to its construction steps.

of the electrical network and creates a graph $G = (N, B)$, where $N = \{n_1, ..., n_k\}$ is the set of nodes of the network and $B = \{b_1, ..., b_k\}$ is the branches connecting the nodes.

The algorithmic complexity of this step is $O(|B|)$, where $|B|$ is the size of the list of dipole equations contained in the input description.

B. Step 2 - Enrichment

Starting from the equations gathered in the previous step and the graph representing the topology of the circuit, this step enriches the list with the application of Kirchhoff's laws. Let KCL and KVL be the set of Kirchhoff's current equations and Kirchhoff's voltage equations, respectively. The algorithm that performs the procedure is shown in Algorithm 1 and an example of resulting hash table is shown in Figure 5. Every equation inside a chain has associated a linked list of dependencies. This conformation therefore allows to partition the set of equations into equivalence classes, where the equation relationship is the linear dependency, thus allowing to disable an entire set of equations if needed.

In the worst case, the introduction of KCL and KVL has a computational complexity of $O(|N|^2)$ and $O(|N|^3)$, respectively. Since the outer loop (lines 4-11) iterates over the set of equations retrieved so far, its complexity is $O(|B|)$. In the worst case, the loop which iterates through all terms of an equation has to consider all the branches, thus resulting in a complexity of $O(|B|)$. The Solve function iterates through all the terms on its right side and, since the longest equation has $|B|$ terms, the complexity of this function is

Program **Input** : The current element of the tree under construction.
Output: The sub-tree which has the current element as root.

```
2  ASSEMBLE(element)  if element == Value then
3      return element;
4  else if element == Variable then
5      if element.isDefined() then
6          if element.hasDerivative() then
7              element = ResolveDerivative(element);
8          return element
9      Equation equation = fetchEquation(element);
10     element.setDefined();
11     element.disable();
12     equation.right = Assemble(equation.right);
13     if equation.hasDerivative() then
14         equation = ResolveDerivative(equation);
15     return equation
16 else if element == Operator then
17     Operator op = copyOperator(element);
18     op.firstOp = Assemble(element.firstOp);
19     op.secondOp = Assemble(element.secondOp);
20     return operator
```

Algorithm 2: Equation tree generation algorithm.

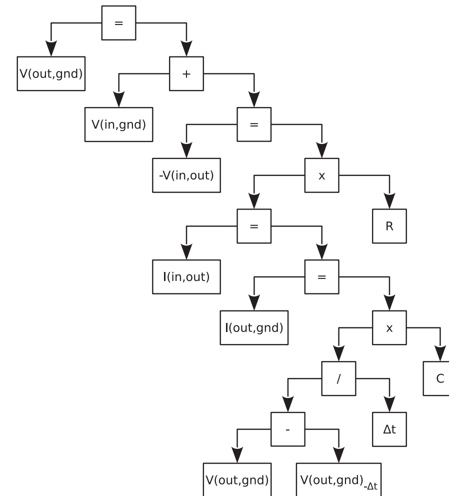

Fig. 6. Final tree representation.

$O(|B|)$. Thus, the algorithmic complexity of Algorithm 1 is $O(|N|^2) + O(|N|^3) + O(|B|^2)$.

C. Step 3 - Assemble

The third step consists in building an intermediate structure, suitable for the next elaboration steps. The algorithm parses the previously generated hash table, and then generates an AST (as detailed in Algorithm 2). This process is exemplified in Figure 6, where the output of interest $V(out, gnd)$ is placed on the top. This recursive algorithm makes use of one equation of each dependency set. Thus, in the worst case, the complexity of this step is linear with the sum of dipole equations and Kirchhoff's equations.

The AST generated from Step 3 represents the linear equation to be solved. Any generated tree has one or more occurrences of the left value on the right side of the equation, as for $V(out, gnd)$ of Figure 6. Having these occurrences on the right-hand side leads to an erroneous description, since

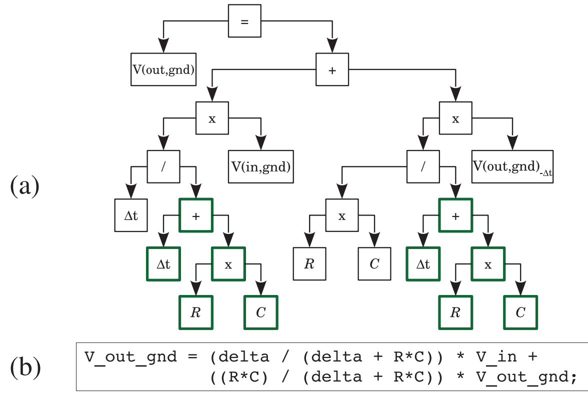

Fig. 7. Solution of the linear equation: (a) the final tree (b) the corresponding generated C++ code.

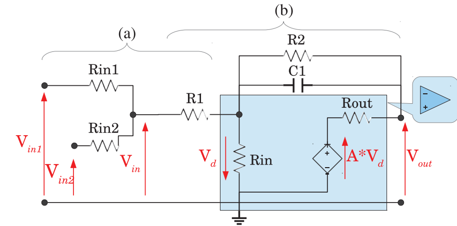

Fig. 8. Electrical diagram of a summing amplifier: (a) two-inputs circuit (b) operational amplifier.

they introduce an unwanted delay due to the interpretation of the symbol equal (which in our case is an assignment). The purpose of this step is thus to remove these occurrences, except in the case where there is the explicit interest on the output value at $-\Delta t$. This allows for the equation to be compatible with the desired description language. Since the language of the final description is C++ (or an extension, e.g., SystemC, SystemC-AMS/TDF), the output of interest appearing on the right side is already delayed by Δt. The algorithm that solves the linear equation is called once the tree is completely formed and before calling the Code Generation algorithm. The result of the application of the algorithm on the tree of Figure 6 is shown in Figure 7.a. In the worst case, the time complexity needed to solve the linear equation, is $O(|N|^3)$.

D. Step 4 - Code Generation and Total Complexity

The purpose of this phase is to generate a description compliant with the selected output language by recursively analysing the given tree. For the tree shown in Figure 7.a, the final code generated by the algorithm is shown in Figure 7.b. The complexity of Step 4 is $O(|B| + |N|)$. If we consider the complexity of all the steps, we will reach a total costs of $O(|N|^3 * |B|^2)$.

V. Experimental Results

This Section shows the effectiveness of the proposed approach by validating the abstraction methodology on single components, and then by showing the impact on the simulation of a complete smart system. Verilog-AMS simulations have been performed by using ELDO, as provided within Questa, while SystemC simulations relied on SystemC-2.3.1 and SystemC-AMS-2.0. Simulation times have been computed by using `clock()` differences for SystemC/C++ descriptions and the ELDO `Global CPU Time` property for Verilog-AMS. Simulations have been performed on a x86_64 architecture, 3.8 GB memory and 2.27×4 GHz CPU, running Linux.

A. Validation of the abstraction methodology

In the first set of experiments, we report accuracy and simulation time of four test cases, i.e., a general n-order RC filter (denoted as RCn and built by cascading n RC stages), a two-inputs circuit (denoted as 2IN and depicted in Figure 8.a) and an operational amplifier (denoted as OA and depicted in Figure 8.b). Circuit parameters for RCn circuits are: R=5kΩ, C = 25nF; for two-inputs circuit are: R1=3kΩ, R2=14kΩ, R3=10kΩ; for the operational amplifier are: R_1=400Ω, R_2=1.6kΩ, C_1=40nF, R_{in}=1MΩ, R_{out}=20Ω The reference model is given in Verilog-AMS. From this model, a SystemC-AMS ELN model has been written manually while the proposed abstraction algorithm has been used to generate SystemC-AMS/TDF, SystemC-DE and C++ versions. All the models are stimulated by a square wave signal generator which is modeled by using the same MoC of the component under test to avoid performance artifacts due to inter-MoCs interfaces. The choice of the square signal has two motivations: first, model inaccuracies are emphasized by transient signals; second, the continuous-time and discrete-time versions of a square wave are almost identical, thus allowing a fair comparison of output signals.

Table I presents the comparison of simulation performances with components in isolation of Verilog-AMS with respect to different SystemC MoCs and C++. The results in Table I are obtained by using a time step of 50 ns, a square wave with a period of 1 ms and 100 ms of simulated time.

The simulation of Verilog-AMS descriptions is the most computationally expensive. Simulation speed increases progressively by removing conservative representation (SystemC-

TABLE I
Simulation performance and accuracy for the abstracted models in isolation.

Component/ Model	Target Language	Generation method	Simulation Time (s)	Error (NRMSE)	Speed-Up
2IN	Verilog-AMS	manual	525.76	0.00	0x
	SC-AMS/ELN	manual	3.15	2.19E-08	167x
	SC-AMS/TDF	algo	2.40	2.41E-08	219x
	SC-DE	algo	1.84	2.41E-08	286x
	C++	algo	0.04	2.41E-08	13144x
RC1	Verilog-AMS	manual	505.95	0.00	0x
	SC-AMS/ELN	manual	2.16	2.10E-09	234x
	SC-AMS/TDF	algo	1.60	4.61E-07	316x
	SC-DE	algo	1.55	4.61E-07	326x
	C++	algo	0.04	4.61E-07	12648x
RC20	Verilog-AMS	manual	596.44	0.00	0x
	SC-AMS/ELN	manual	5.88	4.93E-07	101x
	SC-AMS/TDF	algo	4.16	1.06E-05	143x
	SC-DE	algo	4.21	1.01E-05	141x
	C++	algo	0.14	1.01E-05	4260x
OA	Verilog-AMS	manual	543.23	0.00	0x
	SC-AMS/ELN	manual	2.57	2.44E-07	211x
	SC-AMS/TDF	algo	1.87	1.04E-05	219x
	SC-DE	algo	1.72	1.04E-05	315x
	C++	algo	0.05	1.04E-05	13580x

TABLE II
SIMULATION PERFORMANCE FOR THE ABSTRACTED MODELS, IN ISOLATION, COMPARED TO SYSTEMC-AMS/ELN.

Component /Model	Target Language	Generation Method	Simulation Time (s)	Speed-Up
2IN	SC-AMS/ELN	manual	31.11	0x
	SC-AMS/TDF	algo	25.02	1.24x
	SC-DE	algo	19.00	1.63x
	in C++	algo	0.54	57.61x
RC1	SC-AMS/ELN	manual	21.35	0x
	SC-AMS/TDF	algo	16.27	1.31x
	SC-DE	algo	15.70	1.35x
	C++	algo	0.44	48.52x
RC20	SC-AMS/ELN	manual	60.15	0x
	SC-AMS/TDF	algo	42.99	1.39x
	SC-DE	algo	42.02	1.43x
	C++	algo	1.33	45.22x
OA	SC-AMS/ELN	manual	25.84	0x
	SC-AMS/TDF	algo	19.34	1.33x
	SC-DE	algo	18.51	1.39x
	C++	algo	0.49	52.73x

TABLE III
SIMULATION PERFORMANCE FOR THE ABSTRACTED MODELS INTEGRATED IN THE VIRTUAL PLATFORM.

Component /Model	Component Language	VP Language	Simulator	Gener. Method	Simulation Time(s)	Speed Up (x)
2IN	Verilog-AMS	Verilog	Questa	manual	1067.33	0.00
	Verilog-AMS	SystemC	Questa	manual	729.01	1.46
	SC-AMS/ELN	SystemC	SystemC	manual	57.76	18.47
	SC-AMS/TDF	SystemC	SystemC	algo	54.40	19.62
	SC-DE	SystemC	SystemC	algo	49.19	21.69
	C++	C++	C++	algo	24.62	43.35
RC1	Verilog-AMS	Verilog	Questa	manual	1082.35	0.00
	Verilog-AMS	SystemC	Questa	manual	734.16	1.47
	SC-AMS/ELN	SystemC	SystemC	manual	56.43	19.18
	SC-AMS/TDF	SystemC	SystemC	algo	53.25	20.32
	SC-DE	SystemC	SystemC	algo	48.85	22.15
	C++	C++	C++	algo	26.96	40.14
RC20	Verilog-AMS	Verilog	Questa	manual	1242.29	0.00
	Verilog-AMS	SystemC	Questa	manual	818.94	1.51
	SC-AMS/ELN	SystemC	SystemC	manual	65.91	18.84
	SC-AMS/TDF	SystemC	SystemC	algo	54.22	22.91
	SC-DE	SystemC	SystemC	algo	51.44	24.15
	C++	C++	C++	algo	28.08	44.24
OA	Verilog-AMS	Verilog	Questa	manual	1165.52	0.00
	Verilog-AMS	SystemC	Questa	manual	743.54	1.56
	SC-AMS/ELN	SystemC	SystemC	manual	57.23	20.36
	SC-AMS/TDF	SystemC	SystemC	algo	51.96	22.43
	SC-DE	SystemC	SystemC	algo	50.86	22.91
	C++	C++	C++	algo	27.72	42.04

AMS/TDF), AMS interfaces (SystemC-DE), SystemC scheduler and data types (C++). The equivalence of generated models is evaluated by computing the normalized root-mean-square error (NRMSE) of their output with respect to the output of the original Verilog-AMS representation.

In Table II we removed Verilog-AMS simulation to analyse behavior on a longer simulated time (10 s). The abstraction tool spent 7.67 s to process the most complex model, i.e., RC20 which features 22 nodes and 41 branches. This time is lower than in case of any manual conversion approach.

B. Simulation of the complete smart system

In the second set of experiments, analog components have been simulated in the context of a complete virtual platform for smart systems, as depicted in Figure 1. The digital part consists of a MIPS-based CPU executing assembly instructions contained in the memory, a UART and the APB bus. These digital components are described at RTL. Co-simulation between Verilog-AMS and digital models has been performed by using Questa ADMS by Mentor Graphics. The time step, the period of the square wave and the simulated time are the same as in the previous set of experiments. Results are reported in Table III. Simulation time in Table III are more realistic than Table I because it takes into account the simulation of the whole platform which consists mainly of digital components with only one analog device. The proposed abstraction methodology allows to generate a pure C++ representation which is two times faster than a SystemC-AMS/ELN scenario and 40 times faster than co-simulation with Verilog-AMS.

VI. CONCLUDING REMARKS

This work addressed the problem of efficiently integrating analog devices into virtual platforms to simulate smart systems. Simulation speedup was achieved through an abstraction methodology, that extracts input/output relationships of interest from conservative descriptions and then maps them onto signal flow representations to be written in SystemC-AMS/TDF, SystemC-DE or even C++. Experimental results proved that the generated discrete-event models are 40 times faster than Verilog-AMS co-simulation, and the speedup due to the elimination of the conservative representation is 100%, with a negligible degradation of the output values of interest.

REFERENCES

[1] M. Alassir, J. Denoulet, O. Romain, and P. Garda. Modeling I2C Communication Between SoCs with SystemC-AMS. In *Proc. of IEEE ISIE*, pages 1412–1417, 2007.

[2] P. Benner. Solving large-scale control problems. *Control Systems, IEEE*, 24(1):44–59, Feb 2004.

[3] N. Bombieri, F. Fummi, and G. Pravadelli. Automatic Abstraction of RTL IPs into Equivalent TLM Descriptions. *IEEE TCOMP*, 60(12):1730–1743, 2011.

[4] F. Cenni, O. Guillaume, M. Diaz-Nava, and T. Maehne. SystemC-AMS/MDVP-based modeling for the virtual prototyping of MEMS applications. In *Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP), 2015 Symposium on*, pages 1–6, April 2015.

[5] X. Chen, Y. Wang, and H. Yang. A fast parallel sparse solver for SPICE-based circuit simulators. In *Proc. of DATE*, pages 205–210, 2015.

[6] R. Daniels, H. V. Sosen, and H. Elhak. Accelerating analog simulation with HSPICE precision parallel technology. *Synopsys Tech. Rep.*, 2010.

[7] F. Fummi, M. Lora, F. Stefanni, D. Trachanis, J. Vanhese, and S. Vinco. Moving from co-simulation to simulation for effective smart systems design. In *Proc. of IEEE/ACM DATE*, 2014.

[8] S. Hoelldampf, H. Lee, D. Zaum, M. Olbrich, and E. Barke. Efficient generation of analog circuit models for accelerated mixed-signal simulation. In *IEEE SOC Conference*, pages 104–109, Sept 2012.

[9] Imperas Software. OVP - Open Virtual Platforms. www.ovpworld.org.

[10] Mentor Graphics. Questa Advanced Simulator. www.mentor.com/products/fv/questa.

[11] F. Pecheux, C. Lallemen, and A. Vachoux. VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems. *IEEE TCAD*, 24(2):204–225, Feb 2005.

[12] M. Vasilevski, F. Pecheux, N. Beilleau, H. Aboushady, and K. Einwich. Modeling and Refining Heterogeneous Systems With SystemC-AMS: Application to WSN. In *IEEE/ACM DATE*, pages 134–139, 2008.

[13] S. Vinco, M. Lora, and M. Zwolinski. Conservative behavioural modelling in SystemC-AMS. In *Proc. of FDL*, Sep. 2015.