Analog Models Manipulation for Effective Integration in Smart System Virtual Platforms

Analog Models Manipulation for Effective Integration in Smart System Virtual Platforms

Michele Lora, *Member, IEEE*, Sara Vinco, *Member, IEEE*, Enrico Fraccaroli, *Student Member, IEEE*, Davide Quaglia, *Member, IEEE*, Franco Fummi, *Member, IEEE*

***Abstract*—Analog components are fundamental blocks of smart systems, as they allow a tight interaction with the environment, in terms of both sensing/actuation and communication. This impacts on the design of the overall system, and mainly on the validation phase, that thus requires the joint simulation of digital and analog aspects. In this scenario, this paper proposes the automatic conversion of analog models to C++-based languages, to remove the overhead of co-simulation with traditional virtual platform tools. The proposed methodology allows to convert a given analog description to either (1) a fully equivalent description, or (2) an abstract representation for faster simulation which models only the aspects of interest. Effectiveness and correctness have been proved on a number of case studies, that highlight the effectiveness and potentiality of the proposed methodology.**

***Index Terms*—Heterogeneity, smart systems, analog component simulation, SystemC-AMS, Verilog-AMS, VHDL-AMS, C++ code generation, abstraction**

I. Introduction

Compared to classical embedded systems, a distinctive aspect of smart systems is their *smartness*, *i.e.*, the ability to interact and adapt to an evolving environment, by learning from previous experience and reacting accordingly [35]. This feature makes them a winning solution in a wide range of challenges, spanning across healthcare, factory automation and security, and is mainly enabled by analog components, *i.e.*, sensors and actuators, that allow mutual reaction and sensing between system and environment [36].

The growing importance of the analog domain *w.r.t.* traditional embedded systems has not been compensated by a renewal of the design flows [17]. In fact, embedded SW, digital HW and analog components follow different design flows, targeting custom technologies and techniques that cannot reconcile extremely heterogeneous aspects [15]. As a consequence, no existing framework or language can handle all aspects of a smart system simultaneously [33].

At design time, embedded SW and digital HW are usually integrated through the construction of C++-based virtual platforms, that allow the validation of the HW-SW interaction [9], [19], [32]. Unfortunately, such virtual platforms do not natively support analog descriptions, that are still specified using custom languages, *e.g.* Verilog-AMS, SystemC AMS, and SPICE [1], [2], [26], [28]. Extending the support also to analog components requires the construction of co-simulation frameworks, at the price of an increase of simulation time, that heavily impacts on time-to-market [15], [34].

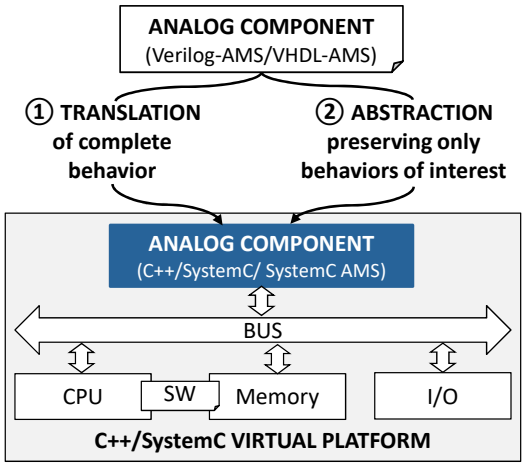


Figure 1: Proposed methodology for simultaneous simulation of analog components with embedded SW and digital HW in virtual platforms.

In this scenario, this paper proposes to enhance the design of smart systems through the *homogeneous and simultaneous simulation of analog components with digital HW and embedded SW*. The adopted strategy consists of converting the starting analog descriptions to C++-based languages. The resulting code can be easily integrated into C++ HW-SW virtual platforms, with no additional co-simulation overhead.

As shown in Figure 1, the proposed methodology supports different levels of adherence *w.r.t.* the starting description, and consequently of simulation performance. If all aspects of the starting description must be preserved the methodology applies a *language translation* (①). Vice versa, if only a subset of the modeled aspects is interesting for the validation of digital HW and SW, the methodology proposes a *model abstraction* flow (②). Both flows lead to the generation of efficient C++-based code, ready to be integrated in the virtual platform.

The main contributions of this work are:

- the definition of a *translation algorithm*, that converts the overall starting description to SystemC AMS. This flow generalizes the methodology in [34]. The algorithm supports only linear models, due to SystemC AMS limitations. Non-linear models may be supported after applying linearization techniques and exploiting the flexibility of SystemC to compose linear models;
- an *abstraction algorithm* supporting both linear and non-linear models, that restricts the initial description to a sub-

M. Lora, E. Fraccaroli, D. Quaglia and F. Fummi is with the Department of Computer Science, University of Verona, Italy, e-mail: name.surname@univr.it.

S. Vinco is with the Department of Control and Computer Engineering, Politecnico di Torino, Italy, e-mail: sara.vinco@polito.it.

set of input/output relations of interest, to achieve faster simulation. This algorithm extends [14] by maximizing the use of symbolic manipulation at generation time, to further remove complexity from simulation;
- the integration of the proposed flows in a *unique sound methodology*, that allows to seamlessly adopt the suitable level of abstraction and to explore the effects of alternative configurations in terms of accuracy and performance;
- the *generation of C++-based code* to be integrated into virtual platforms with no co-simulation overhead;
- the application to *a number of case studies*, that validate the proposed approach on single components, and show the impact on the simulation of a complete smart system.

The paper is organized as follows. Section II provides the necessary background and definitions. Section III presents the overall approach, that is then detailed in Sections IV and V, and applied to experimental case studies in Section VI. Section VII draws our conclusions.

II. State of the art and definitions

A. AMS extensions of hardware description languages

Verilog-AMS and VHDL-AMS present the same modeling concepts, and their differences are mostly syntactic [28]. Thus, even if the following sections adopt the Verilog-AMS syntax, all considerations are applicable to VHDL-AMS as well.

1) Analog and mixed signal management: Verilog-AMS supports descriptions belonging to different physical domains, including electrical, mechanical, and thermodynamics. For this reason, any description must specify the domain and the properties modeled for the system under design.

To this extent, Verilog-AMS defines *natures* (*i.e.*, attributes of the measured quantities, like measure units and absolute tolerance for convergence) and *disciplines*, used to associate system nodes to their measured quantities, that are either potential (*i.e.*, across quantities) or flow (*i.e.*, through quantities) [2]. The most representative discipline in the context of smart systems is the *electrical discipline*, that uses *voltage* as potential (access function `V()`) and *current* as flow (`I()`).

Disciplines associate each system node with both potential and flow natures for *conservative systems*, while *signal-flow* disciplines support only either flow or potential. Equations defined on nodes sharing the same conservative discipline must be in accordance with conservation laws (*e.g.*, a net defined over electrical nodes must obey Kirchhoff's laws).

2) Analog behavior management: The behavior of any system is described as a set of relationships between flows and potentials of nodes and branches (*i.e.*, paths of flows between nodes). These relations are expressed as *contribution statements* (denoted with `<+`), that relate flow and potential quantities of nodes and branches through differential and algebraic equations. Contribution statements allow to infer a system topology. *E.g.*, in case of the electrical discipline, a topology can be inferred whenever the set of relations can be mapped onto a set of basic passive electrical elements (*i.e.*, resistors, capacitors and inductors) and controlled sources (*i.e.*, voltage- or current-controlled sources).

The execution semantics of Verilog-AMS mixes the discrete-event computation typical of HDLs with numerical techniques, necessary to solve continuous-time models. Simulation environments often rely on SPICE-derived solvers [25]. This makes AMS simulation very accurate but slow, thus not allowing an effective simulation of mixed-signal systems [3].

Table I: Taxonomy of analog hardware description levels [29].

Level	Modeling Primitives	Implications
Functional	Mathematical signal flow description per block, connected in signal flow diagram	No internal block structure; conservation laws need not be satisfied on pins
Behavioral	Mathematical description (equations, procedures) per block	No internal block structure; conservation laws must be satisfied on pins
Macromodel	Simplified circuit with controlled sources	Spatially unrelated to actual circuit; conservation laws must be satisfied
Circuit	Connection of SPICE primitives	Spatially one-to-one related to actual circuit; conservation laws must be satisfied

B. SystemC AMS

SystemC AMS extends SystemC with constructs for modeling analog and mixed-signal systems [1]. To cover a wide variety of descriptions, SystemC AMS provides three abstraction levels, supporting different communication styles and representations *w.r.t.* the physical domain. *Electrical Linear Network* (ELN) models electrical networks through the instantiation of predefined primitives, *e.g.*, resistors and capacitors, associated with electrical equations. *Linear Signal Flow* (LSF) adopts signal-flow (*i.e.*, non conservative) representations, but it still supports differential equations. The SystemC AMS internal linear 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. Finally, *Timed Data-Flow* (TDF) models are signal-flow representations, that are scheduled statically by considering their producer-consumer dependencies.

SystemC, both plain [11] and with its AMS extension [10], has been used to model mixed-signal systems. However, none of the previous works provide automatic generation of SystemC AMS modules from previously designed analog models.

C. High level analog modeling and simulation

Behavioral analog modeling is a high-level abstraction of a circuit which describes its behavior as a set of input-output relations. Analog hardware can be described at different levels of abstraction, as shown in Table I. The behavioral level is used both in top-down design flows, *e.g.*, refinement of the circuit from its mathematical behavioral description, as well as in bottom-up verification flows [21].

Even if the design of analog models is tipically top-down [28], recent work proposed bottom-up flows in order to address non-linearities as well as speeding up simulation of analog circuits. In [18] a non-linear analog model is represented as a set of previously-computed linearized versions that are picked during simulation, thus transforming a non-linear model to a set of linear models described at circuit and behavioral level.This approach avoids any numerical integration during simulation but it works only with stepwise

input. [22] extends the previous work by executing an on-the-fly reachability analysis to select only a sub-set of the linearized models. Simulation of non-linear analog circuits is also addressed in [20] by applying a state-space exploration technique. Continuous-time models described as a SPICE netlist are replaced by boolean finite state machines capturing the I/O behavior of the system. However, it requires extensive SPICE simulation in order to extract the behavior. Model Order Reduction (MOR) has been used to achieve faster simulation of analog circuits [4] (both linear [27] and nonlinear circuits [23], [24]) and to reduce complexity of large-scale dynamical models [7] and of multi-physical analog models [31]. However, none of these techniques allow to translate already designed analog models into C++-based languages that may be easily integrated within virtual platforms.

D. Modeling styles

The main classification of modeling styles considered in this work is based on the *adherence to a physical description*. Analog contributions are defined *structural* if they can be mapped onto passive electronic elements (*e.g.*, resistors and capacitors), thus inferring a topology. Otherwise, contributions are called *behavioral*. This definition reflects the behavioral concept as formalized in the digital domain, *i.e.*, a description of a functionality expressed as set of behaviors rather than as an aggregation of sub-components.

E. Formalisms and conventions

Any analog description can be described as a tuple:

$$S =< \mathcal{N}_e, \mathcal{R} >$$

where:

- $\mathcal{N}_e = n_G \cup \{n_i : i \in \mathbb{N}^+\}$: is the set of the *electrical nodes* of the system. By reflecting the Verilog-AMS semantics, this set does not distinguish between internal nodes and interface nodes. $\mathcal{N}_e$ always contains a special node n_G, that represents *ground*.
  From $\mathcal{N}_e$, we derive the set of *electrical branches* $\mathcal{B}_e = \{b_{i,j} = (n_i, n_j) : (n_i, n_j) \in \mathcal{N}_e \times \mathcal{N}_e \wedge n_i \neq n_j\}$. Electrical branches are associated with a current flowing through and an electrical potential across (*i.e.*, voltage). Physical quantities on a branch can be accessed by using the following *access functions*:
  - $\mathtt{V}(b_{i,j})$: voltage on branch $b_{i,j}$, defined as the electric potential difference between nodes n_i and n_j.
  - $\mathtt{I}(b_{i,j})$: amount of current flowing through branch $b_{i,j}$, composed by nodes n_i and n_j.

  Such access functions are generalized through the definition of $\mathtt{P}(b_{i,j})$, that represents any access function for a non-specified physical quantity on branch $b_{i,j}$ (*i.e.*, either $\mathtt{V}(b_{i,j})$ or $\mathtt{I}(b_{i,j})$).
- $\mathcal{R}$: is the set of relations defined by the contribution statements of the model. For electrical linear networks, all contribution patterns can be reduced to:

$$\mathtt{P}_i(b_i) = \left(\sum_{k=1}^{l} C_k \mathtt{P}_k(b_k)\right) + C_{l+1} \qquad (1)$$

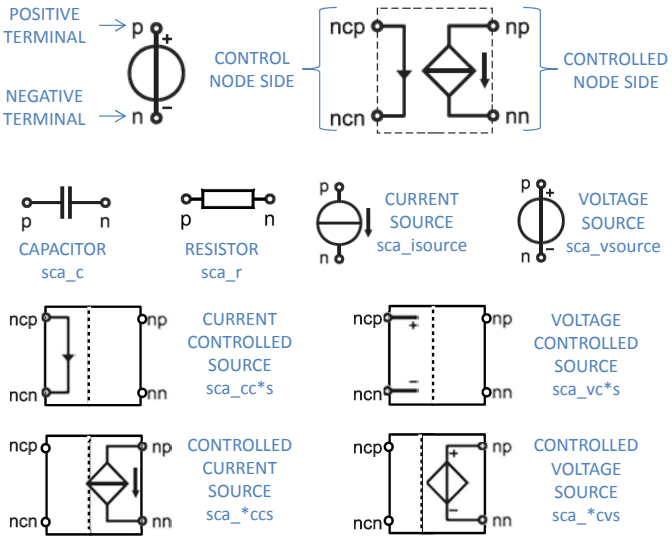


Figure 2: SystemC AMS ELN terminology and main primitives adopted in this work.

$$\mathtt{P}_i(b_i) = C_i * \mathtt{A}\Big(\mathtt{P}_j(b_j)\Big) \qquad (2)$$

where the terms C_i stand for real constants. Operator `A()` generalizes the differential operators `ddt()` (*i.e.*, the derivative operator) and `idt()` (*i.e.*, the integrative operator). `A()`can be applied to any access function.

In the following sections, analog descriptions will be labeled with apices to distinguish between Verilog-AMS (`v`), SystemC AMS (`s`) and C++ implementations (`c`).

It is important to notice that model definitions are based on the set of relations expressed by the model. This allows to reason about models independently from the runtime solver, that may introduce unavoidable numerical errors. As such, different solver-independent *equivalence relations between models* may be defined. In particular, given two systems $S_1 =< \mathcal{N}_1, \mathcal{R}_1 >$ and $S_2 =< \mathcal{N}_2, \mathcal{R}_2 >$ and a mapping function between $\mathcal{N}_1$ and $\mathcal{N}_2$, two possible equivalence relations are defined:

- *node-level equivalence:* S_1 and S_2 are node-level equivalent if $\mathcal{R}_1 = \mathcal{R}_2$ once applied the mapping function between $\mathcal{N}_1$ and $\mathcal{N}_2$;
- *interface-level equivalence:* Let $\mathcal{N'}_1 \subseteq \mathcal{N}_1$ and $\mathcal{N'}_2 \subseteq \mathcal{N}_2$ be the elements of interest for the designer, such that all elements in $\mathcal{N'}_2$ are projections of all elements in $\mathcal{N'}_1$ according to the mapping function from $\mathcal{N'}_1$ to $\mathcal{N'}_2$. Let $\mathcal{R'}_1 \subseteq \mathcal{R}_1$ and $\mathcal{R'}_2 \subseteq \mathcal{R}_2$ be the relations between quantities (*i.e.*, Voltages or Currents) on branches defined respectively over elements of $\mathcal{N'}_1$ and $\mathcal{N'}_2$. S_1 and S_2 are *interface-level equivalent* if $\mathcal{R'}_1 = \mathcal{R'}_2$.

In other words, *node-level equivalence* preserves the relations among all the nodes of the system, while *interface-level equivalence* preserves all the relations between those quantities of the system that are of interest for the designer (usually including all terminals on the component interface).

F. ELN terminology

Top of Figure 2 details the main characteristics of ELN components. ELN modules have a standard interface made up

```
1.  input  in1, in2, in3;
2.  output out;
3.  electrical in, in1, in2, in3, out;
4.  parameter real R1 =   1e+03;
5.  parameter real C1 = 100e-09;
6.  analog begin
7.    V(in)      <+ V(in1)+idt(V(in2)+V(in3))+5.0;
8.    I(in,out)  <+ V(in,out)/R1;
9.    I(out) <+ ddt(V(out))*C1;
10. end
```

Figure 3: Verilog-AMS code of the guiding example.

of a positive terminal (p port) and a negative terminal (n port) for each contributing circuit node (left-hand side). When an ELN module is controlled by any circuit node, the interface has (right-hand side): a source side (*i.e.*, the result of the ELN module, here called the *controlled node*, with a positive terminal np and a negative terminal nn) and a control node side (*i.e.*, the input of the primitive module, here called the *controller node*, with a positive terminal ncp and a negative terminal ncn).

Figure 2 also details the main ELN primitives adopted in this work, as defined by the SystemC AMS standard [1].

G. Guiding Example

Figure 3 shows a synthetic guiding example used throughout the paper. Its representation in terms of a $\mathcal{S}^{\mathsf{v}} =< \mathcal{N}_e^{\mathsf{v}}, \mathcal{R}^{\mathsf{v}} >$ tuple is as follows:

- electrical nodes are:

$$\mathcal{N}_e^{\mathsf{v}} = \{gnd^{\mathsf{v}}, in^{\mathsf{v}}, in1^{\mathsf{v}}, in2^{\mathsf{v}}, in3^{\mathsf{v}}, out^{\mathsf{v}}\}$$

- the only branch explicitly specified is $(in, out) \in \mathcal{B}_e$, other than the (implicit) ones between the nodes and ground;
- contribution statements are represented as relations as follows:

$$\mathcal{R}^{\mathsf{v}} = \left\{ \begin{array}{l} \mathsf{V}(in) = \mathsf{V}(in1) + \int \mathsf{V}(in2) + \mathsf{V}(in3)dt + 5.0, \\ \mathtt{I}(in, out) = \mathsf{V}(in, out)/R1, \\ \mathtt{I}(out) = C1 * d(\mathsf{V}(out))/dt \end{array} \right\}$$

III. Methodology Overview

The proposed methodology for converting analog descriptions into C++-based languages is realized through two techniques, exposing complementary characteristics (as outlined in Figure 1). The main discriminating factor is the desired level of *adherence w.r.t.* the starting description.

Whenever a designer wishes to preserve all behaviors, the code generation process applies a simple *language translation*, by mapping the starting syntactic constructs to SystemC AMS (left-hand side of Figure 1). This preserves all the physical quantities defined on internal nodes of the system, thus producing a model that is *node-level equivalent* to the original. This choice is fundamental when the generated code is the starting point of further analysis or refinements, *e.g.*, to apply power or noise analysis, as well as "white box" verification of internal properties. By referring to Table I, the translation transforms an analog hardware model given at the *circuit level* into a model at the *behavioral level*.

The complementary approach is to focus only on a subset of *behaviors "of interest"*, to speed up simulation. This is achieved through an *abstraction* flow, realized by identifying a sub-set of values of interest of the system (right-hand side of Figure 1): corresponding behaviors are preserved, while any other behavior is pruned. Note that values of interest must be specified by the designer, and they typically carry semantic information necessary to interface the analog component within a mixed-signal environment. They are thus considered as inputs/outputs for the analog device. As a result, the abstraction flow produces a model that is *interface-level equivalent* to the original design, and it moves an analog model from the *circuit* to the *functional level*.

Reducing the starting description to a subset of its possible behaviors is on one hand a limitation, as it restricts the scope of any analysis or "white box" verification. However, this limitation can be overcome by specifying internal values as of values of interest, so that they are preserved during the abstraction process. On the other hand, reducing the starting description is a key advantage, since abstracted models are faster to simulate *w.r.t.* those produced through translation. This *simulation speed-up* is extremely useful when simulating a whole mixed-signal platform to evaluate its global features, and it is achieved without affecting overall system behavior.

The translation and abstraction algorithms differ in terms of supported input models, as highlighted by Table II. Only linear descriptions are supported by both approaches. Translation is constrained to accept only linear models, since the translation algorithm targets SystemC AMS ELN, that relies on a strictly linear solver. The support can be extended to non-linear models only by applying preliminary manipulation to the model, as will be discussed in the next section. On the contrary, the abstraction procedure targets C++ models and performs symbolic resolution through the adoption of solver technologies [5], that nowadays support also non-linear equations. This implies that the scope of application of the abstraction methodology is wider than the scope of translation.

Despite of the differences in the code generation process, the result of both flows can be integrated within C++ or SystemC prototypes in virtual platforms, thus allowing effective evaluation of the heterogeneous system under design.

Table II: Models supported by the proposed approaches.

Methodology	Linear Models	Non-linear Models	
		Piecewise-linear	Algebraic
Translation	✓	(✓)	(~)
Abstraction	✓	✓	✓

IV. Translation Methodology

The translation implements the flow on the left-hand side of Figure 1, and it is represented by a function $\tau(\mathcal{S}^{\mathsf{v}}) = \mathcal{S}^{\mathsf{s}}$, that given a Verilog-AMS implementation $\mathcal{S}^{\mathsf{v}} =< \mathcal{N}_e^{\mathsf{v}}, \mathcal{R}^{\mathsf{v}} >$ returns its node-level equivalent SystemC AMS implementation $\mathcal{S}^{\mathsf{s}} =< \mathcal{N}_e^{\mathsf{s}}, \mathcal{R}^{\mathsf{s}} >$.

Figure 4 gives an overview of the translation procedure. An analog description can be considered as a mean to represent

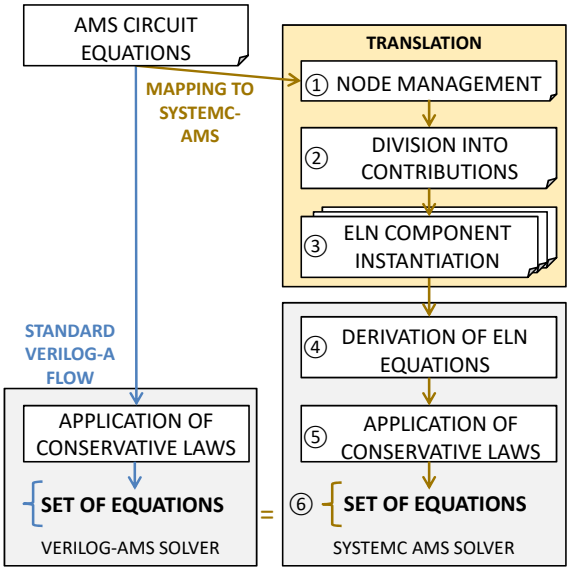


Figure 4: Translation flow for analog component descriptions.

a system of AMS equations composing the circuit. As such, the idea guiding the algorithm is to reproduce the exact set of equations expressed by the analog description (left-hand side of Figure 4) through the instantiation of SystemC AMS ELN primitives (right-hand side of Figure 4).

The translation flow works as follows. Analog nodes are mapped to SystemC AMS nodes (①). Then, every contribution statement is analyzed in order to isolate its basic contributions (②). Each contribution is mapped to ideal ELN primitives, where the equations associated with the ELN primitives are the same as the original contribution (③). The algorithm determines how to connect the ELN modules, so that the bindings describe the same relationships between quantities as the original representation.

The construction of the complete system of equations is demanded to the SystemC AMS internal solver (④), that also takes care of applying conservation laws (⑤). The resulting equations system will thus be node-equivalent to the one described by the starting description, and the resulting model will preserve every detail of the model for what concerns the relations between conservative nodes.

Discussion on supported models: Since the translation algorithm relies on SystemC AMS ELN primitives, only linear descriptions are straightforwardly supported. As anticipated by Table II, support can be extended also to non-linear models with some preliminary maneuver.

Piecewise-linear models may be supported by applying translation to each linear region individually, as proposed by [18]. To ensure that only one region is considered at any simulation instant, regions are wrapped within a control structure composed by SystemC AMS voltage and current sources driven by the Discrete Event SystemC kernel.

Algebraic non-linear models (*e.g.*, models involving polynomials) require to undergo some abstraction, that can be provided either by our abstraction approach, or by state-of-the-art linearization approaches, *e.g.*, [18]. These produced piecewise-linear models that can be treated as above.

Both strategies must be performed at code generation time. Thus, they would not impact simulation performance.

A. Choice of the suitable SystemC AMS abstraction level

A representation obtained through the translation process will not match entirely any abstraction level of SystemC AMS. The generated code is based on the instantiation of an ELN topology composed by ideal components. Thus, it does not represent a physically realizable circuit topology, but rather an aggregation of components reproducing the behavioral relationships between conservative nodes in the original model. As such, the description is considered *behavioral* (see Section II-D). At the same time, the generated code is *conservative*, as ELN primitives predicate over physical quantities of conservative nodes in electrical circuits. Thus, they abide by energy conservation laws (*i.e.*, Kirchhoff's laws). This kind of descriptions constitues a novel modeling formalism in SystemC AMS, called *Analog Behavioral Modeling* (ABM) [34].

The characteristics of ABM models do not fit in any of the SystemC AMS modeling formalisms [34]. However, they are supported by other AMS HDLs and widely used for the design of components such as MEMS and analog circuitry [12], [30]. It is thus necessary to extend SystemC AMS, to improve its coverage and effectiveness. Since the SystemC AMS standard forbids the definition of additional library classes [1], this work proposes an algorithm that maps ABM descriptions to a novel use of SystemC AMS ELN blocks. These blocks are aggregated according to a set of rules guaranteeing to reproduce exactly the set of relations between physical quantities specified in the original model. The use of predefined ELN primitives guarantees the correctness of the underlying synchronization and solving mechanisms, and it preserves compatibility with standard SystemC AMS descriptions.

B. Circuit node management

Step ① of Figure 4 implements the function $\nu(\mathcal{N}_e^{\mathsf{v}}) = \mathcal{N}_e^{\mathsf{s}}$, that maps electrical nodes of the analog implementation into SystemC AMS.

The ground node n_G^v is mapped into node n_G^s, corresponding to an instantiation of a node of type `sca_node_ref`. Any other node n_i^v is mapped into a node n_i^s, that will be declared in SystemC AMS according to the following rules:

- if n_i^v belongs to the interface of the analog model, n_i^s is declared as a `sca_terminal`;
- else, n_i^s is declared as a `sca_node`.

Each node n_i^s inserted into the SystemC AMS implementation (including instances of both `sca_terminal` and `sca_node`) is connected to the ground n_G^s through a 1 GΩ resistor, by using the ELN `sca_r` primitive. This is identical to the *Gmin* conductance inserted by SPICE-based simulators to help convergence.

In the guiding example of Figure 3, the set of SystemC AMS nodes is $\mathcal{N}_e^{\mathsf{s}} = \{gnd^{\mathsf{s}}, in^{\mathsf{s}}, in1^{\mathsf{s}}, in2^{\mathsf{s}}, in3^{\mathsf{s}}, out^{\mathsf{s}}\}$, where gnd is the ground (*i.e.*, `sca_node_ref`), $in1$, $in2$, $in3$ and out are declared as `sca_terminal`, while in is an instance of `sca_node`.

C. Division into contributions

Step ② analyses all the contribution statements and reduces the set of generic relations described by the starting analog

Algorithm 1: Normalization algorithm for the translation procedure.

Input : Initial System (from original specification).
Output: Final Normalized System.

```
1  S' = Normalization(S)
2      S' ← S
3      foreach r in R' do
4          if r ∈ ε + ε2 then
5              (c1, c2) ← Rule1(r)
6              R' ← R' \ {r} ∪ {c1, c2}
7      foreach r in R' do
8          if r ∈ C0 * A(ε) then
9              (c1, c2) ← Rule2(r)
10             R' ← R' \ {r} ∪ {c1, c2}
11     if S' ≠ S then
12         S' ← Normalization(S')
13     return S'
```

implementation into a set of relations expressed in the patterns (1) and (2) (Section II-E).

This pre-processing phase is based onto a set of rules, that divide any original relation into sub-equations. Each couple of sub-equations is connected by an *additional electrical node*, connected to ground by branch b_z. This new node does not have a physical correspondence in the modeled circuit, as it is only used for artificially splitting the described relation. Also this new node is connected to ground through a 1 GΩ resistor.

The following symbols are adopted for the sake of clarity: ϵ_1 to indicate a relation expressed in pattern (1), ϵ_2 for a relation expressed in pattern (2), and ϵ for a generic expression other than a constant, or an access function.

Rule 1 – isolating differential contributions:

$$\mathsf{P}_i(b_i) = \epsilon_1 + \epsilon_2 \rightarrow \begin{cases} \mathsf{P}_i(b_i) = \epsilon_1 + \mathsf{V}(b_z) \\ \mathsf{V}(b_z) = \epsilon_2 \end{cases}$$

Any differential term ϵ_2 contained by the original statement is replaced by the voltage of the new branch b_z. This transformation reduces the original statement in the Form (1). Then, a new contribution in the Form (2) is added, to explicit the equivalence between $\mathsf{V}(b_z)$ and the term ϵ_2.

Rule 2 – managing arguments of differential operators:

$$\mathsf{P}_i(b_i) = C_i * \mathsf{A}(\epsilon) \rightarrow \begin{cases} \mathsf{P}_i(b_i) = C_i * \mathsf{A}(\mathsf{V}(b_z)) \\ \mathsf{V}(b_z) = \epsilon \end{cases}$$

This rule handles all the cases in which the argument of a differential operator is more complex than a single access function. The original argument of the differential operator ϵ is replaced by the voltage of the new branch b_z, thus creating a contribution of type (2). The voltage of b_z is then used as target of a new contribution statement, having as source the original argument of the differential operator (ϵ).

Rule 1 and Rule 2 preserve the relations defined over the branches specified by the original contribution statement. The rules are applied recursively according to Algorithm 1. Given any system S, intended as the set of relations defined over electrical branches, the algorithm returns a normalized equivalent set of relations, expressed only through expressions in patterns (1) and (2).

In detail, for each contribution in the set of relations, the algorithm tries to apply Rule 1 (Lines 3-6). If a contribution is expressed as the trigger condition to apply Rule 1 (Line 4), than the algorithm applies the rule and replaces the original contribution with the new ones (Lines 5–6). Similarly, for each contribution in the resulting set of relations, the algorithm tries to apply Rule 2 (Lines 7-10). Finally, if any modification of the input set occurred, the algorithm is recursively applied to the new set of relations (Lines 11-12). This is necessary because both Rule 1 and Rule 2 may introduce new contributions, that must be normalized. If no modifications occurred, the set S' of relations reached a fixed-point and it can be returned as final result of the normalization.

It is worth noting that Algorithm 1 modifies the input model only by applying Rules 1 and 2. As such, it preservers all the relations over branches specified in the input system.

Guiding example. The following exemplifies the application of Algorithm 1 to the case study in Figure 3. Given the system $\mathcal{S}^{\mathsf{v}} =< \mathcal{N}_e^{\mathsf{v}}, \mathcal{R}^{\mathsf{v}} >$, line 2 creates a new system $\mathcal{S}' =< \mathcal{N}_e', \mathcal{R}' >$ where $\mathcal{N}_e' = \mathcal{N}_e^{\mathsf{v}} = \{gnd, in, in1, in2, in3, out\}$,

$$\mathcal{R}' = \mathcal{R}^{\mathsf{v}} = \left\{ \begin{array}{l} \mathsf{V}(in) = \mathsf{V}(in1) + \int \mathsf{V}(in2) + \mathsf{V}(in3)dt + 5, \\ \mathtt{I}(in, out) = \mathsf{V}(in, out)/R1, \\ \mathsf{V}(out) = C1 * d(\mathsf{V}(out))/dt \end{array} \right\}$$

Since the first relation in $\mathcal{R}'$ is in the form $\epsilon + \epsilon_2$ (Line 4), the algorithm applies Rule 1 (Line 5), thus adding a new node $n1$ in $\mathcal{N}_e'$ and modifying the set $\mathcal{R}'$ as follows (Line 6):

$$\mathcal{R}' = \left\{ \begin{array}{l} \mathsf{V}(n1) = \int \mathsf{V}(in2) + \mathsf{V}(in3)dt, \\ \mathsf{V}(in) = \mathsf{V}(in1) + \mathsf{V}(n1) + 5, \\ \mathtt{I}(in, out) = \mathsf{V}(in, out)/R1, \\ \mathsf{V}(out) = C1 * d(\mathsf{V}(out))/dt \end{array} \right\}$$

The newly introduced relation is in the form $C_0 * \mathsf{A}(\epsilon)$ (Line 8), hence Rule 2 can be applied (Line 9), thus adding node $n2$ in $\mathcal{N}_e'$ and modifying $\mathcal{R}'$ as follows:

$$\mathcal{R}' = \left\{ \begin{array}{l} \mathsf{V}(n2) = \mathsf{V}(in2) + \mathsf{V}(in3), \\ \mathsf{V}(n1) = \int \mathsf{V}(n2)dt, \\ \mathsf{V}(in) = \mathsf{V}(in1) + \mathsf{V}(n1) + 5, \\ \mathtt{I}(in, out) = \mathsf{V}(in, out)/R1, \\ \mathsf{V}(out) = C1 * d(\mathsf{V}(out))/dt \end{array} \right\}$$

Since $\mathcal{S}' \neq \mathcal{S}$, the function is recursively applied to $\mathcal{S}'$. However, no further transformation is performed, and the normalized system $\mathcal{S}'$ is returned. △

D. ELN Components instantiation

Step ③ recreates the normalized relations produced by Algorithm 1 by instantiating and connecting ELN components. The procedure differs for the two formats of contributions. In the following, figures follow a chromatic convention: light blue for current and red for voltage, while yellow portions are dependent on the type of contribution to reproduce.

Type (1) contributions: This rule applies to all contribution statements of pattern (1):

$$P_i(b_i) = \left(\sum_{k=1}^{l} C_k \mathsf{P}_k(b_k) \right) + C_{l+1}$$

Note that, since we are dealing with linear systems, it is sufficient to take care of the addition of physical values. Figure 5 exemplifies the instantiation for the relation:

$$\mathsf{P}_0(n_1, n_2) = \mathsf{P}_1(n_3, n_4) + \mathsf{P}_2(n_5, n_6) + ... + \mathsf{P}_3(n_{k-1}, n_k) + C$$

The sum is implemented as parallel composition of controlled current sources (left-hand side of Figure 5). The control side of such sources (in yellow) depends on the operands appearing on the right-hand side of the contribution statement. For any k, if P_k is `V()`, then the instantiated component is a voltage-controlled current source (*i.e.*, `sca_vccs`). If else P_k is `I()`, the instantiated component is a current-controlled current source (*i.e.*, `sca_cccs`). The positive terminal of the control interface is connected to branch b_k, and the gain of the controlled source is set to C_k.

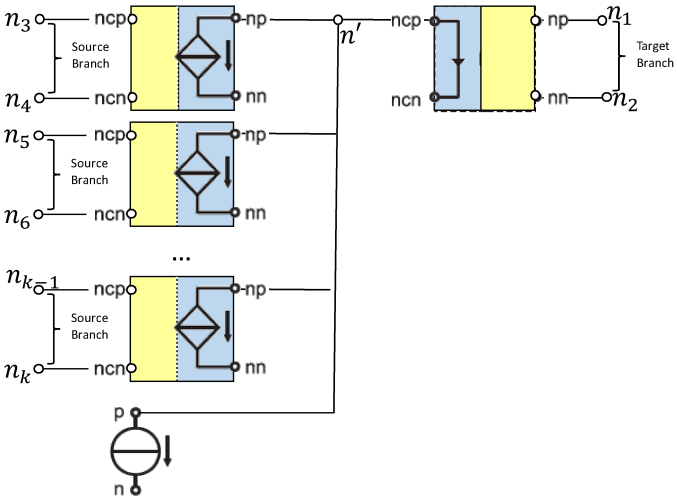


Figure 5: Topological pattern for Type (1) contributions.

All current sources are connected in parallel to a new intermediate node n', that is connected to the control side of a current controlled source (right-hand side of Figure 5). The control side of this block (in yellow) once again depends on the starting contribution. If the target of the contribution $P_i()$ is `V()`, the block is a voltage source. Else if the target of the contribution $P_i()$ is `I()`, the block is a current source. The controlled interface is then connected to the target branch b_i.

Finally, the constant value C_{l+1} in the summation is reproduced by connecting in parallel a constant current source, whose generated current is equal to the value of C_{l+1}.

Let us consider the topology instantiated by the described algorithm from a contribution of type (1). The algorithm assures that the equations solved by the SystemC AMS solver are equivalent to the ones in the original contribution. The current entering the node n' is equal to the sum of the right-hand operands of the original contribution statement. Simply applying the Kirchhoff Current Law (KCL), this is also equal to the current flowing in the branch $(n', ground)$. Thus, it will be the output quantity generated by the current controlled source connected to the target branch.

Type (2) contributions: Differential contributions are more complex, as they model a derivative (or integrative) relationship between currents or voltages of two separate circuit branches. SystemC AMS, on the other hand, restricts differential behaviors to dependencies on single network branches, through the adoption of capacitors or inductors. (`sca_l` ELN primitive). To overcome this limitation, it is necessary to introduce an intermediate node that has no physical correspondence in the circuit, but that is rather used for describing the differential dependence.

All the differential contributions are mapped using the generic topological pattern depicted in Figure 6.

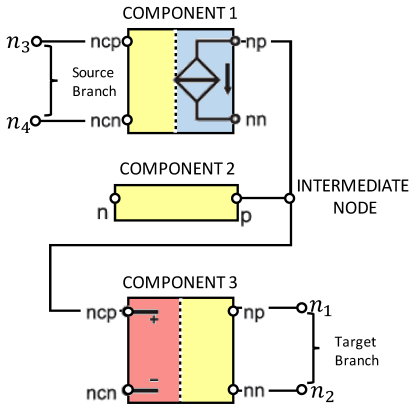


Figure 6: Topological pattern for contributions in Form (2).

Component 1 is a controlled current source, as indicated by the controlled side (in blue). The control side (in yellow) depends on the modeled contribution. If the argument of the derivative construct is `V()`, Component 1 is a voltage-controlled current source (*i.e.*, `sca_vccs`). Else, if the argument is `I()`, Component 1 is a current-controlled current source (*i.e.*, `sca_cccs`).

Component 3 is a voltage-controlled source, as indicated by the control side (in red). The controlled side (in yellow) reflects the target of the Verilog-AMS contribution statement. If the target is `V()`, Component 3 is a voltage controlled voltage source (*i.e.*, `sca_vcvs`). Else if the target is `I()`, Component 3 is a voltage-controlled current source (*i.e.*, `sca_vccs`).

Component 2 (in yellow in Figure 6) is used to create the differential relation between the current value, controlled by Component 1, and the voltage value controlling Component 3. Component 2 is an inductor whenever the differential contribution is derivative (*i.e.*, `sca_l`), and it is a capacitor in case of an integrative contribution (*i.e.*, `sca_c`). Thus, given $I_{np,nn}$ the current flowing through terminals `np` and `nn` of Component 1, and $V_{ncp,ncn}$ the voltage on the branch between the terminals `ncp` and `ncn` of Component 3, the relationship described by Component 2 is:

$$V_{ncp,ncn} = \int I_{np,nn} dt$$

in case of a derivative contribution (*i.e.*, Component 2 is a capacitor), and

$$V_{ncp,ncn} = \frac{dI_{np,nn}}{dt}$$

in case of an integrative contribution (*i.e.*, Component 2 is an inductor).

Considering the derivative and the integrative operators of Verilog-AMS, we can describe all possible configurations in terms of the eight cases in Table III. For each case, the table shows the corresponding SystemC AMS primitives.

Table III: Summary of the components employed to map differential contributions.

Contribution	Component 1	Component 2	Component 3
`I(`n_1, n_2`)<+k ddt(I(`n_3, n_4`))`	Current Controlled Current Source `sca_cccs`	Inductor `sca_l`	Voltage Controlled Current Source `sca_vccs`
`I(`n_1, n_2`)<+k ddt(V(`n_3, n_4`))`	Voltage Controlled Current Source `sca_vccs`	Inductor `sca_l`	Voltage Controlled Current Source `sca_vccs`
`V(`n_1, n_2`)<+k ddt(I(`n_3, n_4`))`	Current Controlled Current Source `sca_cccs`	Inductor `sca_l`	Voltage Controlled Voltage Source `sca_vcvs`
`V(`n_1, n_2`)<+k ddt(V(`n_3, n_4`))`	Voltage Controlled Current Source `sca_vccs`	Inductor `sca_l`	Voltage Controlled Voltage Source `sca_vcvs`
`I(`n_1, n_2`)<+k idt(I(`n_3, n_4`))`	Current Controlled Current Source `sca_cccs`	Capacitor `sca_c`	Voltage Controlled Current Source `sca_vccs`
`I(`n_1, n_2`)<+k idt(V(`n_3, n_4`))`	Voltage Controlled Current Source `sca_vccs`	Capacitor `sca_c`	Voltage Controlled Current Source `sca_vccs`
`V(`n_1, n_2`)<+k idt(I(`n_3, n_4`))`	Current Controlled Current Source `sca_cccs`	Capacitor `sca_c`	Voltage Controlled Voltage Source `sca_vcvs`
`V(`n_1, n_2`)<+k idt(V(`n_3, n_4`))`	Voltage Controlled Current Source `sca_vccs`	Capacitor `sca_c`	Voltage Controlled Voltage Source `sca_vcvs`

Given a contribution of Type (2), the set of equations defined by the topology instantiated as described is equivalent to the original contribution. Let us consider a contribution of Type (2), where A is a derivative operator `ddt`. Given the additional node n', its physical quantities are defined as:

$$\mathtt{I}(n', n_G) = \mathtt{P}_j(b_j) \qquad \mathtt{V}(n', n_G) = \mathtt{P}_i(b_i)$$

The algorithm adds the equation for the inductor connecting n' and n_G with inductance value C_i:

$$\mathtt{V}(n', n_G) = C_i * \frac{dI(n', n_G)}{dt}$$

By replacing the values, we obtain:

$$\mathtt{P}_i(b_i) = C_0 * \frac{d\mathtt{P}_j(b_j)}{dt}$$

Let us consider now a contribution of type (2) where A is an integrative operator `idt`. The physical quantities on the intermediate node n' are:

$$\mathtt{I}(n', n_G) = \mathtt{P}_j(b_j) \qquad \mathtt{V}(n', n_G) = \mathtt{P}_i(b_i)$$

The algorithm adds a capacitor with capacity value of C_i, connecting n' and n_G:

$$\mathtt{V}(n', n_G) = C_i * \int \mathtt{I}(n', n_G) dt$$

Thus, replacing the values:

$$\mathtt{P}_i(b_i) = C_i * \int \mathtt{P}_j(b_j) dt$$

Finally, it is possible to conclude that the topology generated by applying the instantiation rules is *node-level* equivalent to the input model of the translation algorithm.

Guiding example. Figure 7 depicts the topology of components instantiated by applying the algorithm to the Verilog-AMS model in Figure 3. For the sake of readability, disconnected terminals in the figure are intended as connected to the ground node. White nodes (*i.e.*, $in1$, $in2$, $in3$, in and out) are the ones explicitly specified in the original model. Yellow nodes (*i.e.*, $n1$ and $n2$) are nodes inserted during node management (Section IV-B). Green "unnamed" nodes are inserted during ELN component instantiation to connect basic blocks of the topology (Section IV-D). Note that all nodes are connected to ground via the 1 GΩ resistors. △

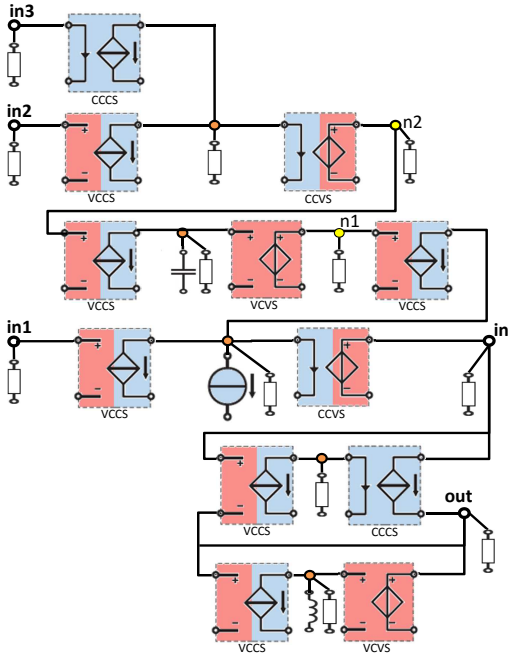


Figure 7: Resulting topology obtained by the translation of the guiding example in Figure 3.

E. Complexity

The complexity of the proposed translation algorithm is derived from its constituting steps:

- *Step* ①: the instantiation of the SystemC AMS nodes and $1G\Omega$ resistors is performed in constant time for every node. Thus the complexity for this step is $O(|\mathcal{N}_e|)$.
- *Step* ②: the application of Rules 1 and 2 is constant. Thus the complexity of the step is the complexity of Algorithm 1. Its worst case happens when every contribution statement is of maximum length (*i.e.*, $O(|\mathcal{N}_e|^2)$) and every branch appears on the left-hand side of a contribution statement (*i.e.*, $O(|\mathcal{N}_e|^2)$). Thus, the complexity of this step is $O(|\mathcal{N}_e|^2) \cdot O(|\mathcal{N}_e|^2) = O(|\mathcal{N}_e|^4)$.
- *Step* ③: a topological pattern is instantiated for every addend in any relation generated after the previous step. The maximum number of addends per relation is $O(|\mathcal{N}_e|^2)$, while the relations are at most $O(|\mathcal{N}_e|^2)$. Thus,

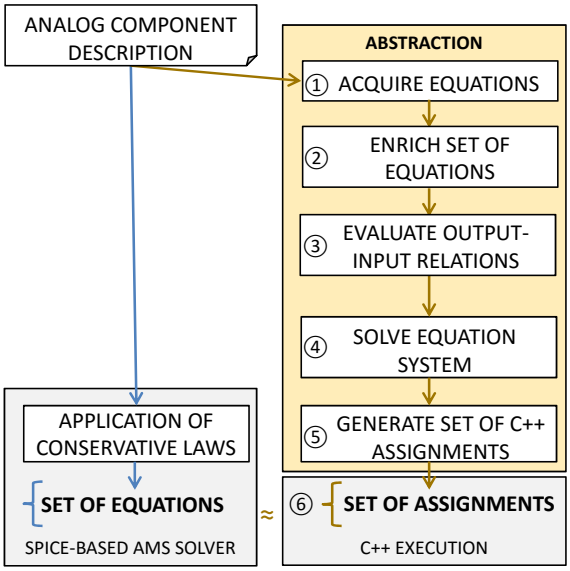


Figure 8: Abstraction flow for analog component descriptions.

the complexity of this step is $O(|\mathcal{N}_e|^4)$.

In conclusion, the total complexity of the translation procedure is the sum of these three steps:

$$O(|\mathcal{N}_e|) + O(|\mathcal{N}_e|^4) + O(|\mathcal{N}_e|^4) = O(|\mathcal{N}_e|^4)$$

V. Abstraction Methodology

The abstraction flow can be represented by the function $\alpha(\mathcal{S}^{\mathsf{v}}, \mathsf{P}(n)) = \mathcal{S}^{\mathsf{c}}$. Given a Verilog-AMS implementation $\mathcal{S}^{\mathsf{v}} =< \mathcal{N}_e^{\mathsf{v}}, \mathcal{R}^{\mathsf{v}} >$ and a value of interest $\mathsf{P}(n)$, $\alpha(\mathcal{S}^{\mathsf{v}}, \mathsf{P}(n))$ returns a C++ model $\mathcal{S}^{\mathsf{c}} =< \mathcal{N}_e^{\mathsf{c}}, \mathcal{R}^{\mathsf{c}} >$, such that any relation between input values and the quantity $\mathsf{P}(n)$ is preserved. Since α deals with only one value of interest $\mathsf{P}(n)$, it is applied to $\mathcal{S}^{\mathsf{v}}$ once for each value of interest to be preserved.

Figure 8 gives an overview of the abstraction approach. The guiding idea is to analyze the starting analog description to restrict the model to the sub-equations binding the specified value of interest to the inputs of the model. Note that the starting description can be both linear and non-linear.

The first step of the algorithm generates new equations by replacing the left-hand side of each equation with the terms on its right-hand side (①). As a next step, circuit topology is inferred to extend the system of equations with the application of Kirchhoff's conservation laws (②). Then, the algorithm analyses the whole set of equations to identify the sub-set describing the relation between the specified value of interest and the inputs of the model (③). The sub-set of equations is then solved by means of a symbolic solver, with the goal of breaking all the algebraic loops (④). The result is used to generate the behavioral C++ description (⑤).

A. Circuit equations acquisition

Step ① parses all relations in $\mathcal{R}^{\mathsf{v}}$ and translates each of them into an abstract syntax tree (AST). In each AST, leaves represent values and variables, while intermediate nodes represent operators. Each element of the tree is associated with a number of flags for storing additional information, *e.g.*, the presence of derivative or integrative operators. The generated equations are then stored inside a Multimap, *i.e.*, an efficient data structure which requires $O(1)$ to insert an element, and a worst case effort proportional to the list length $O(l)$ to search and delete an element.

The translation to ASTs is based on five rules, divided into Left-Hand Side rules (LHS) and Right-Hand Side rules (RHS). In the remainder of this section, for each relation $r_i \in \mathcal{R}^{\mathsf{v}}$, ϵ_i identifies the expression on its right-hand side.

LHS Rule 1: Given any branch $b_{i,j}$, if $\mathcal{R}^{\mathsf{v}}$ contains one and only one relation r_k containing an access function $P(b_{i,j})$ on the LHS (*i.e.*, the LHS defines a quantity of branch $b_{i,j}$), then the access function is replaced by a variable as follows:

$$P(b_{i,j}) = \epsilon_k \rightarrow P\_i\_j = \epsilon_k$$

LHS Rule 2: Given any branch $b_{i,j}$, we define $\mathcal{R}_{par}$ as the set of relations having a current access function $I(b_{i,j})$ on the LHS. For each $r_k \in \mathcal{R}_{par}$, with $1 \leq k \leq |\mathcal{R}_{par}|$, the LHS of r_k is replaced by a variable as follows:

$$I_k(b_{i,j}) = \epsilon_k \rightarrow I\_i\_j\_k = \epsilon_k$$

Suffix k is necessary because multiple relations may assign a current value over the same pair of nodes. Such relations actually make an assignment over distinct *parallel* branches, and thus must be treated separately. Adding k as variable suffix allows to preserve the distinction between different branches.

LHS Rule 3: Given any branch $b_{i,j}$, we define $\mathcal{R}_{ser}$ as the set of relations having a voltage access function $V(b_{i,j})$ on the LHS. The management of $\mathcal{R}_{ser}$ introduces a set N_s of intermediate nodes, with $|N_s| = (|\mathcal{R}_{ser}| - 1)$. For each $r_k \in \mathcal{R}_{ser}$, with $1 \leq k \leq |\mathcal{R}_{ser}|$, the LHS of r_k is replaced by a variable as follows:

$$V_k(b_{i,j}) = \epsilon_k \rightarrow \begin{cases} V\_i\_n_k = \epsilon_k & \text{if } k = 1 \\ V\_n_{(k-1)}\_n_k = \epsilon_k & \text{if } 1 < k < |\mathcal{R}_{ser}| \\ V\_n_{(k-1)}\_j = \epsilon_k & \text{if } k = |\mathcal{R}_{ser}| \end{cases}$$

where n_k and n_{k-1} are respectively the k-th intermediate node and its precedent. Note that the intermediate nodes are introduced because relations assigning a value to voltage over the same pair of nodes actually refer to branches *in series*. LHS Rule 3 preserves this by distributing the relations onto distinct pairs of nodes, included in $\{n_i\} \cup N_s \cup \{n_j\}$.

RHS Rule 1: A relation containing a differential operator over a sum of access functions on the RHS is modified moving the operator from the entire expression to its single elements:

$$\mathsf{A}\left(\sum_{k=1}^{l} C_k \mathsf{P}_k(b_k)\right) \rightarrow \sum_{k=1}^{l} \left(C_k * \mathsf{A}(\mathsf{P}_k(b_k))\right)$$

RHS Rule 2: Any access function is replaced with a variable as follows:

$$P(b_{i,j}) \rightarrow P\_i\_j$$

Guiding example. Considering the case study in Figure 3 as its formalization $\mathcal{S}^{\mathsf{v}} =< \mathcal{N}_e^{\mathsf{v}}, \mathcal{R}^{\mathsf{v}} >$, where:

$$\mathcal{R}^{\mathsf{v}} = \left\{ \begin{array}{l} \mathsf{V}(in) = \mathsf{V}(in1) + \int(\mathsf{V}(in2) + \mathsf{V}(in3))dt + 5, \\ \mathtt{I}(in, out) = \mathsf{V}(in, out)/R1, \\ \mathtt{I}(out) = d(\mathsf{V}(out))/dt * C1 \end{array} \right\}$$

Set		Equations
$\mathcal{R}^{\mathsf{c}}$	Z	$V\_in = V\_in1 + \int(V\_in2)dt + \int(V\_in3)dt + 5$
	A	$I\_in\_out = V\_in\_out/R1$
	B	$I\_out = d(V\_out)/dt * C1$
$\mathcal{R}^{\mathsf{c}}_{\mathsf{d}}$	C	$V\_in\_out = I\_in\_out * R1$
	D	$V\_out = \int(I\_out)dt/C1$
$\mathcal{R}^{\mathsf{c}}_{\mathsf{kvl}}$	E	$V\_in\_out = V\_in - V\_out$
	F	$V\_out = V\_in - V\_in\_out$
$\mathcal{R}^{\mathsf{c}}_{\mathsf{kcl}}$	G	$I\_in\_out = I\_out$
	H	$I\_out = I\_in\_out$

Table IV: Equations gathered (A, B and Z) and generated (from C to H) by the abstraction procedure.

The LHS and RHS rules lead to the definition of a new set of relations $\mathcal{R}^{\mathsf{c}}$:

$$\mathcal{R}^{\mathsf{c}} = \left\{ \begin{array}{l} V\_in = V\_in1 + \int(V\_in2)dt + \int(V\_in3)dt + 5 \\ I\_in\_out = V\_in\_out/R1 \\ I\_out = d(V\_out)/dt * C1 \end{array} \right\}$$

△

B. Equation system enrichment

The second step of the abstraction approach infers circuit topology to enrich the set of relations $\mathcal{R}^{\mathsf{c}}$ with Kirchhoff's conservation laws. The starting point consists of *partitioning the set of relations* into two sub-sets: $\mathcal{R}^{\mathsf{c}}_{\mathsf{bhv}}$, containing behavioral equations, and $\mathcal{R}^{\mathsf{c}}_{\mathsf{str}}$, devoted to structural equations:

$$\mathcal{R}^{\mathsf{c}}_{\mathsf{bhv}} = \left\{ V\_in = V\_in1 + \int(V\_in2)dt + \int(V\_in3)dt + 5 \right\}$$

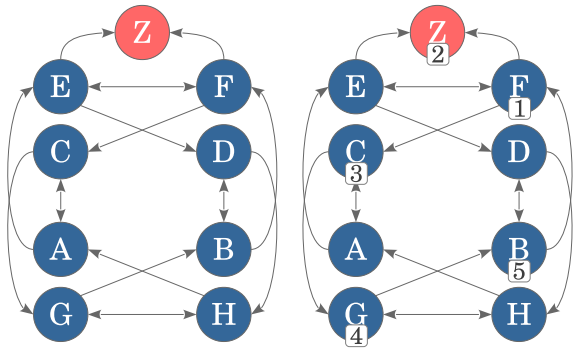
$$\mathcal{R}^{\mathsf{c}}_{\mathsf{str}} = \left\{ \begin{array}{l} I\_in\_out = V\_in\_out/R1 \\ I\_out = d(V\_out)/dt * C1 \end{array} \right\}$$

$$\mathcal{R}^{\mathsf{c}} = \mathcal{R}^{\mathsf{c}}_{\mathsf{bhv}} \bigcup \mathcal{R}^{\mathsf{c}}_{\mathsf{str}}$$

Table IV exemplifies the enrichment step application on the guiding example, and is used as a reference throughout the remainder of this section. Equations A, B and Z are the ones derived from the previous step.

The construction of circuit topology insists only on the sub-set of structural equations $\mathcal{R}^{\mathsf{c}}_{\mathsf{str}}$, due to their adherence to a physical description. *Kirchhoff's conservation laws* [13] allow to derive two new sets of equations: $\mathcal{R}^{\mathsf{c}}_{\mathsf{kcl}}$ and $\mathcal{R}^{\mathsf{c}}_{\mathsf{kvl}}$. $\mathcal{R}^{\mathsf{c}}_{\mathsf{kcl}}$ contains those equations derived from $\mathcal{R}^{\mathsf{c}}$ through the application of Kirchhoff's Current Law (KCL), and it contains $|\mathcal{N}^{\mathsf{v}}_e| - 1$ independent equations. Complementarily, $\mathcal{R}^{\mathsf{c}}_{\mathsf{kvl}}$ contains $|\mathcal{B}^{\mathsf{v}}_e| - |\mathcal{N}^{\mathsf{v}}_e| + 1$ independent equations derived by using Kirchhoff's Voltage Law (KVL). Considering the guiding example, this process leads to the definition of Equations E, F, G and H of Table IV. Note that equations describing the same circuit mesh (*i.e.*, E and F) are linerly dependent. The same relation applies to the equations describing currents entering and exiting the same node (*i.e.*, G and H).

The next step *derives the dual relation of each equation* in $\mathcal{R}^{\mathsf{c}}_{\mathsf{str}}$ [6], by interchanging the relation left-hand side term with the right-hand side terms. This introduces a new set of equations $\mathcal{R}^{\mathsf{c}}_{\mathsf{d}}$ linearly dependent from the original one. The size of the new set of equations is $|\mathcal{R}^{\mathsf{c}}_{\mathsf{d}}| = |\mathcal{R}^{\mathsf{c}}_{\mathsf{str}}|$. The application of this step is exemplified by Equations C and D in Table IV, which are duals *w.r.t.* Equations A and B.



(a) (b)

Figure 9: Dependencies graph generated for the guiding example starting from the equations of Table IV (a), and application of graph visit (b).

Linear dependencies between equations are stored in a linked-list, *i.e.*, each equation inside a set of linearly dependent equations points to the next equation belonging to the same set. This allows to partition the set of equations into equivalence classes based on the linear dependency relation.

C. Cone of influence exploration

The third step of the methodology determines the equations necessary to describe the outputs of the model *w.r.t.* its inputs. To ease the application of this step, relations between equations are represented by a *graph of dependencies*, like the one in Figure 9a. The graph is built as follows:

- *Node Creation Rule*: Each equation introduces a node, labeled with the corresponding LHS variable.
- *Edge Creation Rule*: An edge connects nodes n_i and n_j whenever the LHS variable associated to the node n_i appears on the RHS of the equation associated to n_j.

For the sake of readability, both equations in Table IV and in Figure 9 are labeled with letters. Consider node H in Figure 9a, associated with equation H of Table IV, *i.e.*, $I\_out = I\_in\_out$. The RHS of the equation contains the variable I_in_out, that is also used on the LHS of the equations represented by nodes A and G. This adds to the graph an edge from H to A, and an edge from H to G.

The graph is visited according to the following rules:

- *Node visit rule*: When a node n_i is visited, the node is disabled and the corresponding equation e_i is stored inside a set called $\mathcal{R}_{es}$. Then, all nodes representing equations linearly dependent *w.r.t.* e_i are disabled.
- *Disabled node rule*: Disabled nodes cannot be visited.
- *Next node rule*: After completing the visit of a node n_i, the visit moves to all non disabled neighbors of n_i. If all nodes of the graph are disabled the visit ends.

After the visit, $\mathcal{R}_{es}$ is the smallest set of equations describing the relation between system inputs and the values of interest.

Guiding example. Figure 9b depicts an example of exploration of the graph in Figure 9a starting from node F.

Numbers represent the order in which the nodes are visited. The resulting set of equations is:

$$\mathcal{R}_{es} = \begin{cases} V\_out = V\_in - V\_in\_out \\ V\_in = V\_in1 + \int(V\_in2)dt + \int(V\_in3)dt + 5 \\ V\_in\_out = I\_in\_out * R1 \\ I\_in\_out = I\_out \\ I\_out = d(V\_out)/dt * C1 \end{cases} \quad (3)$$

△

D. Equations system solver

Algebraic loops may lead to an erroneous simulation if not properly managed. The fourth step of the methodology aims at removing them by solving the equations system.

The first step deals with *time-dependent operators* (*i.e.*, `ddt` and `idt`), that are not managed by symbolic solvers. Each equation is visited and occurrences of the aforementioned operators are replaced with the corresponding discrete-time approximation. Different techniques of numerical differentiation or integration can be used, depending on the desired degree of accuracy. A simple example of approximation technique for the derivative operator is the finite difference formula in Equation 4, where h is the simulation step of the model:

$$\frac{dV(t)}{dt} \longrightarrow \frac{V(t) - V(t-1)}{h} \quad (4)$$

For the integrative operator, a typical example is the quadrature formula in Equation 5, where variable V_{acc} is an accumulator incremented by $(V(t) * h)$ at the end of each simulation step:

$$\int V(t)dt \longrightarrow (V(t) * h) + V_{acc} \quad (5)$$

Note that replacing all the time-dependent operators with the corresponding approximation formula moves the model semantics from continuous- to discrete-time, with timestep h.

To ensure the correctness of the final code, equations must then be solved by a symbolic solver capable of dealing with systems of linear equations. In this work, the choice fell on GiNaC [5], a *C++ library for symbolic computation*. Given a system of linear equations and its unknown variables, the symbolic engine provides a function called `lsolve`, which solves the system and returns a set of equations describing the *functional behavior* of the electrical model. This set can be mapped onto C++-based descriptions, ranging from pure C++ to SystemC AMS TDF. These generated models can be easily integrated into C++-based virtual platforms.

Guiding example. The result of applying the proposed approximations to Equation 3 is:

$$\mathcal{R}'_{eq} = \begin{cases} V\_in = V\_in1 \\ \qquad + (V\_in2 * h) + V\_in2\_acc \\ \qquad + (V\_in3 * h) + V\_in3\_acc + 5 \\ V\_out = V\_in - V\_in\_out \\ V\_in\_out = I\_in\_out * R1 \\ I\_in\_out = I\_out \\ I\_out = ((V\_out - V\_out\_prev)/h) * C1 \end{cases} \quad (6)$$

Figure 10 depicts then the corresponding C++ code, obtained through GiNaC. △

```
1. double f(double Vin1, double Vin2, double Vin3)
2. {
3.     V_in = V_in1 + (V_in2*h) + V_in2_acc
4.                  + (V_in3*h) + V_in3_acc + 5.0;
5.     V_out=(C1*R1*V_out_prev + V_in*h)/(C1*R1+h);
6.     // Update variables.
6.     V_out_prev = V_out;
7.     V_in2_acc += V_in2 * h;
8.     V_in3_acc += V_in3 * h;
9.     return V2;
10. }
```

Figure 10: Abstracted C++ description generated for the guiding example in Figure 3.

E. Complexity

In the worst case scenario, the initial set of relations $\mathcal{R}^{\mathsf{v}}$ contains only structural equations. As consequence, the dimension of $\mathcal{R}^{\mathsf{v}}$ is at most $|\mathcal{N}_e|^2$.

The complexity of the proposed abstraction approach is derived from its constituting steps:

- *Step* ①: Access function renaming and the integral (derivative) decomposition is constant for each equation in $\mathcal{R}^{\mathsf{v}}$. The algorithmic complexity of this step is given by the size of the set: $O(|\mathcal{N}_e|^2)$.
- *Step* ②: Gathering KCL and KVL equations has a worst case running time respectively of at most $O(|\mathcal{N}_e|^2)$ and $O(|\mathcal{N}_e|^3)$. Dual relation generation is linear *w.r.t.* the length of each structural equation, that is constant, and it must be applied to at most $O(|\mathcal{N}_e|^2)$ equations, thus resulting in a complexity of $O(|\mathcal{N}_e|^2)$. Thus, the overall complexity of this step is $O(|\mathcal{N}_e|^2) + O(|\mathcal{N}_e|^3) + O(|\mathcal{N}_e|^2) = O(|\mathcal{N}_e|^3)$.
- *Step* ③: Equations are partitioned into equivalence classes, whose number is given by the dimension of initial set of relations and the number of independent Kirchhoff's equations. During graph exploration, these classes are disabled whenever a node associated with one of their equations is visited. Therefore, the graph exploration complexity is $O(|\mathcal{N}_e|^2) + (O(|\mathcal{N}_e|) - 1) + (O(|\mathcal{N}_e|^2) - O(|\mathcal{N}_e|) + 1) = O(|\mathcal{N}_e|^2)$.
- *Step* ④: The number of equations selected by the exploration is at most equal to the number of branches in the circuit. Solving an equation system of dimension $|\mathcal{N}_e|^2$ has a computational complexity of $O(|\mathcal{N}_e|^6)$ [5].
- *Step* ⑤: Generating the set of C++ assignments requires at most $O(|\mathcal{N}_e|^2)$.

The overall computational complexity is thus:

$$O(|\mathcal{N}_e|^2) + O(|\mathcal{N}_e|^3) + O(|\mathcal{N}_e|^6) = O(|\mathcal{N}_e|^6)$$

VI. Experimental results

This section proves the effectiveness of the proposed methodology on a number of case studies of increasing complexity. All experiments have been executed on a 64-bit machine running Ubuntu 14.04, equipped with 16 GB of memory and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz.

All proposed steps have been automated in the ASTRAL tool, *i.e.*, the *Analog Systems Translation and absRAction tooL*. ASTRAL relies on the HIFSuite framework [8] for parsing and manipulation of Verilog-AMS models.

Table V: Benchmarks characteristics and generation time.

Benchmark	Relations	Values of Interest		LoC	Abstraction Time (s)	Translation Time (s)
		Input	Output			
RC1	2	1	1	17	0.026	0.041
IN2	3	2	2	21	0.029	0.051
PIFilter	4	1	1	21	0.035	0.075
IN3	5	3	2	31	0.037	0.058
Op-Amplifier	6	1	1	46	0.036	0.064
RC5	10	1	1	42	0.069	0.108
RC10	20	1	1	67	0.195	0.197
RC20	40	1	1	117	0.828	0.411
Accelerometer	66	10	8	123	0.417	0.402

A. Case studies

The case studies used for the experimental analysis are:

- four low-pass filters with an increasing number of stages (*i.e.*, *RC1*, *RC5*, *RC10* and *RC20*);
- a capacitor-input filter (*i.e.*, *PIFilter*), composed by a couple of capacitors, an inductor and a load resistance;
- two multi-input circuits, composed by the interconnection of passive basic electrical components, with two (*i.e.*, *IN2*) and three (*i.e.*, *IN3*) inputs respectively;
- an operational amplifier (*i.e.*, *Op-Amplifier*);
- an accelerometer, modeled using a set of algebraic differential equations expressing behavioral relations over electrical values.

Table V reports the characteristics of the benchmarks, in terms of number of relations, number of selected values of interest, and lines of code (LoC) of the starting Verilog-AMS model. The adopted case studies have an increasing number of relations, to prove the scalability of the proposed methodology.

B. Methodology accuracy

The accuracy of the proposed approach is estimated by comparing a reference Simulink model of each benchmark *w.r.t.* both the original Verilog-AMS description (simulated by using SPICE) and the code generated through translation and abstraction. Table VI reports the accuracy estimated for the four low-pass filters, by showing the *normalized root-mean-square error* (NRMSE) on the computed outputs. The low error rate of both the abstracted and translated codes highlights the high level of accuracy of the generated models. The NRMSE rages from 10^{-6} to 10^{-7}, that is comparable to the precision obtained using SPICE-based simulators.

Table VI: NRMSE of the generated models *w.r.t.* Simulink.

Description	RC1	RC5	RC10	RC20
SPICE	2.81E-07	2.92E-07	7.28E-07	3.60E-07
Translation	1.84E-06	1.83E-07	2.13E-06	1.75E-06
Abstraction	8.14E-07	9.55E-07	1.65E-06	1.47E-06

C. Methodology performance

The performance of the generated code is evaluated by considering all benchmark individually.

The *generation time* (reported in columns *Translation time (s)* and *Abstraction time (s)* of Table V) is always well below 1s, for all benchmarks and code versions. Whenever the model is mostly composed by structural equations (*e.g.*, *RC20*), the abstraction flow requires more time than the translation flow. On the contrary, models comprising primarily behavioral equations (*e.g.*, *Accelerometer*) speed up the abstraction flow, since steps ② and ③ are not applied.

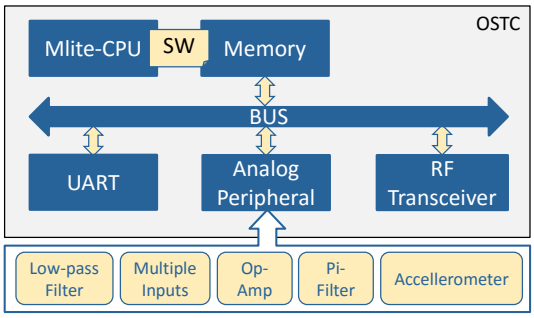


Figure 11: Structure of the OSTC virtual platform.

To estimate the *simulation performance*, we considered three scenarios for each benchmark:

- Verilog-AMS description, simulated with Questa [25];
- SystemC AMS ELN code, generated through translation;
- C++ code, generated through the abstraction flow.

Each scenario executes 1 second of simulated time with a fixed time step of 50 nanoseconds. The adoption of a fixed time step is necessary to ensure correct interaction of the analog benchmarks with the digital sub-system [16]. The fixed time step degrades SPICE simulation speed *w.r.t.* adaptive step simulation, as the simulator has to re-evaluate the overall analog sub-system more often.

The resulting simulation times are reported in columns *Component time (s)* of Table VII. The speed-up achieved for the different case studies depends on their internal structure. Nonetheless, both the translation and the abstraction models proved to improve simulation time for all benchmarks. Translation achieves a maximum speed-up of 67.0x, with an average speed-up of 37.4x. Abstraction further fastens simulation by reaching 2 orders of magnitude speed-ups (ranging from 711.0x to 122.1x), for an average speed-up of 335.7x.

D. Application to a smart system scenario

To prove the effectiveness of the generated code in the context of virtual platforms, the generated models have been integrated in a mixed-signal virtual platform: the Open Source Test Case (OSTC) [15], available as open-source demonstrator for HIFSuite. The structure of the OSTC is depicted in Figure 11: it comprises a SW application running on top of a general-purpose CPU, and a number of both digital and analog peripherals which communicate through a bus. The Verilog-AMS models of the analog components are integrated and simulated within a mixed VHDL, Verilog and SystemC version of the OSTC. The SystemC AMS and C++ versions of the analog components are integrated and simulated within a C++ implementation of the OSTC. Each implementation of the platform is stimulated with the same testbench carrying on a transient analysis covering 1 second of simulated time, with a time step of 50 nanoseconds.

Table VII reports the *execution time* required to simulate the platform including the benchmarks in each code version (columns *Platform time (s)*). The speed-up achieved for the

Table VII: Execution times for different abstractions simulated both alone and together with the smart system.

Benchmark	Heterogeneous		Translation – SystemC AMS/ELN				Abstraction – C++			
	Component	Platform	Component		Platform		Component		Platform	
	Time (s)	Time (s)	Time (s)	Speed-up (x)	Time (s)	Speed-up (x)	Time (s)	Speed-up (x)	Time (s)	Speed-up (x)
RC1	4,898.45	8,751.36	73.16	67.0	539.38	16.2	6.89	711.0	70.79	123.6
IN2	3,706.86	7,358.91	86.88	42.7	568.31	13.0	11.69	317.1	70.95	103.7
PIFilter	5,097.50	8,905.17	99.83	51.1	569.54	15.6	9.75	522.8	71.52	124.5
IN3	3,815.47	7,465.37	114.73	33.3	623.40	12.0	16.75	227.8	71.51	104.4
Op-Amplifier	5,174.22	6,369.19	105.45	49.1	559.73	11.4	19.63	263.6	71.24	89.4
RC5	5,307.23	9,075.30	151.58	35.0	612.57	14.8	12.56	422.6	73.55	123.4
RC10	6,152.12	9,826.31	268.72	22.9	722.17	13.6	22.59	272.3	82.13	119.6
RC20	7,746.65	11,288.23	443.50	17.5	939.35	12.0	63.45	122.1	111.93	100.9
Accelerometer	7,749.64	12,388.71	576.73	13.4	1,348.39	9.2	47.83	162.0	82.84	149.6

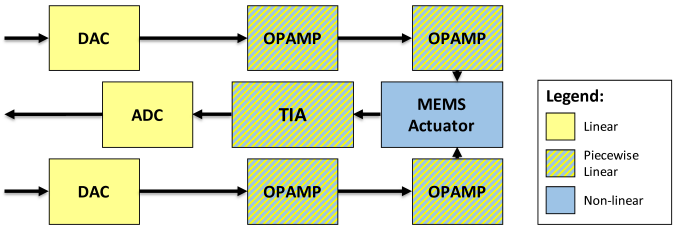


Figure 12: Structure of the non-linear case study.

simulation of the component in isolation is mitigated in this scenario by the execution of the remainder of the platform. However, the achieved speed-up is always one order of magnitude for the translation flow (maximum 16.2x, average 13.09x), and two orders of magnitude for the abstraction flow (ranging between 89.4x and 124.5x, average 115.4x).

Even in this case simulation times show a certain variability among the test cases. The speed-up achieved by simulating a component, may not be completely reached once the component is integrated within the platform: *e.g.*, IN3 and RC20 reach the same speed-up, but IN3 is faster once integrated in the OSTC. This is caused by the overhead introduced to manage the component interface and the communication with the other components composing the platform.

Note that the methodology proved to improve simulation performance in every considered scenario and for all case studies. Thus, it allows to effectively and efficiently simulate an entire virtual platform of a smart system, still guaranteeing negligible accuracy losses.

E. Non-linear case study

This section focuses on the analysis of the applicability of the proposed approaches to a system presenting non-linearities. The case study (depicted in Figure 12) contains:

- three linear sub-components: two Digital-to-Analog Converters (DAC), one Analog-to-Digital Converter (ADC);
- five Piecewise linear components: four Operational Amplifier (OPAMP) and a Trans-Impedance Amplifier (TIA);
- a MEMS Actuator, whose behavior is described by using polynomial functions.

Table VIII reports the application to the non-linear case study of the same analysis carried on for linear benchmarks in Section VI-D. The *Heterogeneous* line reports the simulation time of the original Verilog-AMS description, simulated both as a single component and within the OSTC.

Table VIII: Simulation results on the non-linear case study.

Code Version	Component		Platform	
	time(s)	speed-up(x)	time(s)	speed-up(x)
Heterogeneous	9,067.63	–	10,156.27	–
Translation	385.57	23.51	661.78	15.34
Abstraction	112.43	80.65	276.62	36.71

The second line (*i.e.*, *Translation*) reports the time needed to simulate a SystemC model of the device, as obtained through translation. Given the presence of non-linear components, translation could be applied straightforwardly only to linear and piecewise-linear sub-components. Piecewise-linear sub-components additionally required the introduction of wrappers for managing discontinuities, and the non-linear sub-component has been replaced by its abstracted version, wrapped with a SystemC interface. The last line of Table VIII (*i.e.*, *Abstraction*) reports the results of abstraction, that has been straightforwardly applied to the entire device.

The table highlights that simulation performance worsens w.r.t. the linear benchmarks. When applying translation, simulation is slowed down by the introduction of computationally expensive interfaces, necessary to handle discontinuities. In the case of the abstraction, the overhead is caused by both the introduction of SystemC wrappers and by the increased complexity of the assignments, that include polynomials and other burdening numerical operations. Still, the proposed methodology enhances simulation speed *w.r.t.* heterogeneous simulation, thus proving the effectiveness of the generated code and that the proposed methodology could be successfully applied also to quite complex non-linear devices.

VII. Concluding Remarks

This paper proposed a methodology for integrating analog descriptions in virtual prototypes for smart systems. The methodology provides two alternative flows with different characteristics in terms of adherence *w.r.t.* the starting description and of simulation speed-up. Effectiveness and correctness have been shown on a number of case studies, as well as on a smart system prototype. Experimental results highlight the effectiveness, efficiency and correctness of the approach, thanks to a worst case NRMSE in the order of 10^{-6} and to a maximum speed-up of 711.0x.

References

[1] Accellera. SystemC-AMS Language Reference Manual. 2014.

[2] Accellera. Verilog-AMS Language Reference Manual. 2014.
[3] M. Alassir, J. Denoulet, O. Romain, and P. Garda. Modeling I2C Communication Between SoCs with SystemC-AMS. In *Proc. of IEEE ISIE 2007*, pages 1412–1417.
[4] H. Aridhi, M. H. Zaki, and S. Tahar. Towards improving simulation of analog circuits using model order reduction. In *Proc. of the ACM/IEEE DATE 2012*, pages 1337–1342.
[5] C. Bauer, A. Frink, and R. Kreckel. Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *Elsevier JSC*, 33(1):1 – 12, 2002.
[6] V. Belevitch. Summary of the History of Circuit Theory. *Proc. of the IRE*, 50(5):848–855, 1962.
[7] P. Benner. Solving large-scale control problems. *IEEE Control Systems*, 24(1):44–59, 2004.
[8] N. Bombieri, M. Ferrari, F. Fummi, et al. HIFSuite: tools for HDL code conversion and manipulation. *EURASIP JES*, 2010(1):1–20, 2010.
[9] Cadence. Virtual System Platform.
[10] F. Cenni, S. Scotti, and E. Simeu. Behavioral modeling of a CMOS video sensor platform using SystemC AMS/TLM. In *Proc. of IEEE/ECSI FDL 2011*, pages 1–6.
[11] Z. Chen, Y. Wang, L. Liao, Y. Zhang, A. Aytac, J. H. Muller, R. Wunderlich, and S. Heinen. A SystemC Virtual Prototyping based methodology for multi-standard SoC functional verification. In *Proc. of IEEE/ACM DAC 2014*, pages 1–6. IEEE.
[12] Coventor, Inc. MEMS+: MEMS Simulation Software. 2011.
[13] P. Feldmann and R. Rohrer. Proof of the number of independent Kirchhoff equations in an electrical circuit. *IEEE TCAS*, 38(7):681–684, 1991.
[14] E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi. Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems. In *Proc. of IEEE/ACM DATE 2016*, pages 1–6.
[15] 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*, pages 1–4.
[16] G. G. Gielen and R. A. Rutenbar. Computer-aided design of analog and mixed-signal integrated circuits. *Proceedings of the IEEE*, 88(12):1825–1854, 2000.
[17] M. Grosso, F. Cenni, G. Gangemi, S. Rinaudo, M. Crepaldi, A. Sanginario, and D. Demarchi. Enabling Smart System design with the SMAC Platform. In *Proc. of IEEE DTIP 2015*, pages 1–6.
[18] S. Hoelldampf, H. Lee, D. Zaum, M. Olbrich, and E. Barke. Efficient generation of analog circuit models for accelerated mixed-signal simulation. In *Proc. of IEEE SOCC 2012*, pages 104–109.
[19] Imperas Software. OVP - Open Virtual Platforms.
[20] A. V. Karthik, S. Ray, P. Nuzzo, A. Mishchenko, R. Brayton, and J. Roychowdhury. ABCD-NL: Approximating continuous non-linear dynamical systems using purely boolean models for analog/mixed-signal verification. In *Proc. of ACM/IEEE ASP-DAC 2014*, pages 250–255.
[21] L. Lavagno, L. Scheffer, and G. Martin. *EDA for IC implementation, circuit design, and process technology*. CRC press, 2006.
[22] H. S. L. Lee, M. Althoff, S. Hoelldampf, M. Olbrich, and E. Barke. Automated generation of hybrid system models for reachability analysis of nonlinear analog circuits. In *Proc. of ACM/IEEE ASP-DAC 2015*, pages 725–730.
[23] P. Li and L. T. Pileggi. Compact reduced-order modeling of weakly nonlinear analog and RF circuits. *IEEE TCAD*, 24(2):184–203, 2005.
[24] H. Liu, L. Daniel, and N. Wong. Model reduction and simulation of nonlinear circuits via tensor decomposition. *IEEE TCAD*, 34(7):1059–1069, 2015.
[25] Mentor Graphics. Questa Advanced Simulator.
[26] L. W. Nagel and D. O. Pederson. *SPICE: Simulation program with integrated circuit emphasis*. Electronics Research Laboratory, College of Engineering, University of California, 1973.
[27] A. Odabasioglu, M. Celik, and L. T. Pileggi. PRIMA: Passive reduced-order interconnect macromodeling algorithm. In *Proc. of IEEE/ACM DAC 1997*, pages 58–65.
[28] F. Pêcheux, C. Lallement, and A. Vachoux. VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems. *IEEE TCAD*, 24(2), 2005.
[29] L. Scheffer, L. Lavagno, and G. Martin. *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC Taylor & Francis, 2006.
[30] P. Schneider, C. Bayer, K. Einwich, and A. Kohler. System level simulation - A core method for efficient design of MEMS and mechatronic systems. In *Proc. of IEEE SSD 2012*, pages 1–6.
[31] R. Sommer, T. Halfmann, and J. Broz. Automated behavioral modeling and analytical model-order reduction by application of symbolic circuit analysis for multi-physical systems. *Elsevier Simulation Modelling Practice and Theory*, 16(8):1024–1039, 2008.
[32] Synopsys. Platform Architect.
[33] S. Vinco, M. Lora, V. Guarnieri, J. Vanhese, D. Trachanis, and F. Fummi. Design Domains and Abstraction Levels for Effective Smart System Simulation. In *Smart Systems Integration and Simulation*, chapter 3, pages 23–54. Springer, 2016.
[34] S. Vinco, M. Lora, and M. Zwolinski. Conservative Behavioural Modelling in SystemC-AMS. In *Proc. of IEEE/ECSI FDL 2015*.
[35] S. Vinco and C. Pilato. Editorial: Special Issue on Innovative Design Methods for Smart Embedded Systems. *ACM TECS*, 15(2), 2016.
[36] V.K.Varadan, K. Vinoy, and S. Gopalakrishnan. *Smart material systems and MEMS: design and development strategies*. John Wiley and Sons, 2006.

Michele Lora (M'13) received the Ph.D. degree in computer science from the University of Verona, Italy, in 2016. He currently is a Post-Doctoral Research Associate at the Department of Computer Science, University of Verona, Italy. His research focuses on automation methodologies for modeling, integration and efficient simulation of heterogeneous embedded systems and contract-based requirement engineering for cyber-physical systems.

Sara Vinco (M'09) received the Ph.D. degree in computer science from the University of Verona, Italy, in 2013. She is currently a Post-Doctoral Research Associate at the Department of Control and Computer Engineering, Politecnico di Torino, Italy. Her main research interests are energy efficient electronic design automation and techniques for simulation and validation of heterogeneous embedded systems.

Enrico Fraccaroli (S'16) is a...

Davide Quaglia (M'03) received his Ph.D. in Computer Engineering from Politecnico di Torino (Italy) in 2003. Currently he is Assistant Professor at the Computer Science Department of the University of Verona (Italy). He is member of IEEE and Euromicro DSD Program Committee. He is author/co-author of about 70 papers about Networked Embedded Systems, Networked Control Systems, Cyber-Physical Systems. He is also co-founder and active project leader of EDALab, a spin-off company of the University of Verona.

Franco Fummi (M'92) received the Ph.D. degree in electronic engineering from Politecnico di Milano, Italy, in 1995. He is currently the Head of the Department of Computer Science, University of Verona, Italy, where he is a Full Professor since 2000, and where he became an Associate Professor in computer architecture in 1998. Since 1995, he has been with the Department of Electronics and Information, Politecnico di Milano, as an Assistant Professor. He is a co-founder of EDALab, an EDA company developing tools for the design of networked embedded systems. His current research interests include electronic design automation methodologies for modeling, verification, testing, and optimization of embedded systems.