UNIVERSITY OF VERONA

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE SCHOOL OF NATURAL SCIENCES AND ENGINEERING

DOCTORAL PROGRAM IN COMPUTER SCIENCE

CYCLE XXXI

A Holistic Approach to Functional Safety for Networked Cyber-Physical Systems

S.S.D. ING-INF/05

Coordinator: Prof. Massimo Merro

Tutor: Prof. Franco Fummi

Doctoral Student: Dott. Enrico Fraccaroli

Abstract

Functional safety is a significant concern in today's networked cyber-physical systems such as connected machines, autonomous vehicles, and intelligent environments. Simulation is a well-known methodology for the assessment of functional safety. Simulation models of networked cyber-physical systems are very heterogeneous relying on digital hardware, analog hardware, and network domains. Current functional safety assessment is mainly focused on digital hardware failures while minor attention is devoted to analog hardware and not at all to the interconnecting network.

In this work we believe that in networked cyber-physical systems, the dependability must be verified not only for the nodes in isolation but also by taking into account their interaction through the communication channel. For this reason, this work proposes a holistic methodology for simulation-based safety assessment in which safety mechanisms are tested in a simulation environment reproducing the high-level behavior of digital hardware, analog hardware, and network communication. The methodology relies on three main automatic processes: 1) abstraction of analog models to transform them into system-level descriptions, 2) synthesis of network infrastructures to combine multiple cyber-physical systems, and 3) multi-domain fault injection in digital, analog, and network.

Ultimately, the flow produces a homogeneous optimized description written in C++ for fast and reliable simulation which can have many applications. The focus of this thesis is performing extensive fault simulation and evaluating different functional safety metrics, *e.g.*, fault and diagnostic coverage of all the safety mechanisms.

Abstract (Italian)

Al giorno d'oggi la sicurezza funzionale è uno degli aspetti più interessanti dei sistemi cyber-fisici di rete, come macchinari interconnessi, veicoli autonomi e ambienti intelligenti. La simulazione è una tecnica ben nota per la valutazione della sicurezza funzionale. Purtroppo, i modelli utilizzati per simulare sistemi cyber-fisici di rete sono molto eterogenei e si basano su hardware digitale, hardware analogico e aspetti di rete. Il metodo con cui attualmente si valuta la sicurezza funzionale si concentra principalmente sui guasti dell'hardware digitale, mentre minore attenzione è dedicata all'hardware analogico e quasi nessun interesse per la rete che interconnette tutte le piattaforme.

La convinzione su cui si basa questo lavoro è che nei sistemi cyber-fisici di rete, l'affidabilità deve essere verificata non solo per i nodi in isolamento, ma anche tenendo conto della loro interazione attraverso il canale di comunicazione. Per questo motivo, questo lavoro propone una metodologia olistica per la valutazione della sicurezza basata sulla simulazione in cui i meccanismi di sicurezza sono testati in un ambiente di simulazione che riproduce il comportamento ad alto livello dell'hardware digitale, dell'hardware analogico e della comunicazione di rete. La metodologia si basa interamente su tre processi automatizzati: 1) astrazione di modelli analogici per trasformarli in descrizioni system-level, 2) sintesi di infrastrutture di rete per unire assieme multipli sistemi cyber-fisici, e 3) iniezione di guasti multi dominio in digitale, analogico e rete.

In definitiva, il flusso produce una descrizione ottimizzata omogenea scritta in C++ la quale permette una simulazione veloce e affidabile con molte applicazioni. Il focus di questa tesi è l'esecuzione di una simulazione di guasti e la valutazione di diverse metriche di sicurezza funzionale, ad esempio, *fault* e *diagnostic coverage* di tutti i meccanismi di sicurezza implementati sulle piattaforme di rete.

Acknowledgements

In primo luogo, vorrei esprimere la mia sincera gratitudine al mio relatore, il Prof. Franco Fummi per l'incrollabile sostegno durante il mio dottorato. I suoi consigli e la sua saggezza mi hanno guidato nella mia ricerca e mi hanno reso il ricercatore che sono oggi. Anche quando i suoi consigli parevamo portarmi a navigare senza una bussola o senza le stelle come guida, invero mi stavano indirizzando verso lidi ignoti pieni di nuove ed interessanti scoperte. Questa è l'essenza di un dottorato, esplorare senza paura e nel frattempo migliorare noi stessi.

In secondo luogo vorrei ringraziare il Prof. Davide Quaglia che per primo a fatto nascere in me la curiosità per i miei studi. Nel corso della mia laurea magistrale e durante le sue lezioni, inconsapevolmente mi convinse ad intraprendere il dottorato e mi trasmise la sua incredibile curiosità che principalmente lo caratterizza. Sia il Prof. Fummi che il Prof. Quagli sono stati un grande esempio di ricercatore per me, e cercherò sempre di far tesoro degli innumerevoli (non scherzo) consigli che mi hanno dato durante il dottorato.

Vorrei ringraziare il Prof. Riccardo Muradore del mio comitato di tesi, per i suoi preziosi e costruttivi suggerimenti, commenti e incoraggiamenti. Lo ringrazio inoltre per la sua curiosità che mi ha spinto ad ampliare la mia ricerca ed ad osservarla sotto varie prospettive.

Vorrei esprimere la mia profonda gratitudine alla Dott. Renaud Gillon, che mi ha dato l'opportunità di unirmi al suo team come stagista. Mi ha ripetutamente fornito un supporto prezioso e ha pazientemente risposto alle mie innumerevoli domande. Nell'ultimo anno e mezzo mi ha insegnato moltissime cose, e lo ha fatto con un incredibile entusiasmo. Già prima di incontrarlo, trasmettere le mie conoscenze era un'attività piacevole e entusiasmante, ora è diventata una passione.

Devo ringraziare i miei colleghi per le profonde, stimolanti, e utili discussioni che riguardavano tutto tranne che il lavoro. Li ringrazio per aver condiviso con me innumerevoli scadenze (queste vanno per prime), eventi, e incredibili banchetti a conferenze in terre lontane. Inoltre li ringrazio per aver sopportato i miei numerosi colloqui con gli studenti, uno dei tanti momenti esilaranti di un dottorato.

Ultimo ma non meno importante, vorrei ringraziare tutta la mia famiglia a partire dai miei genitori e mio fratello. Loro mi hanno spinto a seguire quello in cui credevo e sopratutto mi hanno insegnato a credere in me stesso. Vorrei inoltre ringraziare la mia fidanzata per l'incrollabile pazienza e supporto durante il mio dottorato. Durante un dottorato ci sono molti momenti difficili, avere qualcuno con cui poterne parlare è incredibilmente importante.

Grazie a tutti voi,

Enrico

Contents

List of Figures

List of Tables

List of Listings

Introduction

1.1 Introduction

Electronics, microsystems and cyber-physical systems underpin innovation and value creation across the economy. Especially in the areas of networked cyber-physical systems and smart systems composed of heterogeneous components such as analog sensors, MEMS, power sources, digital hardware, network interfaces, embedded software, and so on. A more ad more critical aspect in the design of such systems is related to the safety-critical domains. As a consequence, functional safety assessment of such devices and ensuring their dependability are becoming sophisticated and critical tasks. Current approaches analyze the different domain of a smart system with separated methodologies. Thus, they are not able to measure the cross interaction between analog, digital and network components and to correlate them to extra-functional properties like power-consumption, thermal dissipation, and aging.

Simulation is a well-known and widely-used methodology for the assessment of functional safety. However, simulation models of networked cyber-physical systems are very heterogeneous relying on digital hardware, analog hardware, and network domains. Current functional safety assessment is mainly focused on digital hardware failures while minor attention is devoted to analog hardware and not at all to the interconnecting network. This is a problem which must be solved in the nearest feature, especially now that recent trends are pushing in the direction of Internet-of-Things (IoT) devices, where everyday objects are being connected to the Internet. Companies could design such devices by using off-the-shelf components, however relying on custom System-on-Chips (SoCs) instead would give them more control over the balance between size, power, and cost. Based on these trends, the interest for Networked Cyber-Physical System (NCPS) will defensively arise in the foreseeable, and so the interest in more efficient design flows.

This work proposes a holistic methodology for simulation-based safety assessment in which safety mechanisms are tested in a simulation environment reproducing the high-level behavior of digital hardware, analog hardware, and network.

1.2 Methodology flow

Figure 1.1 shows the overall structure of the proposed methodology, where two flows meet in the middle.

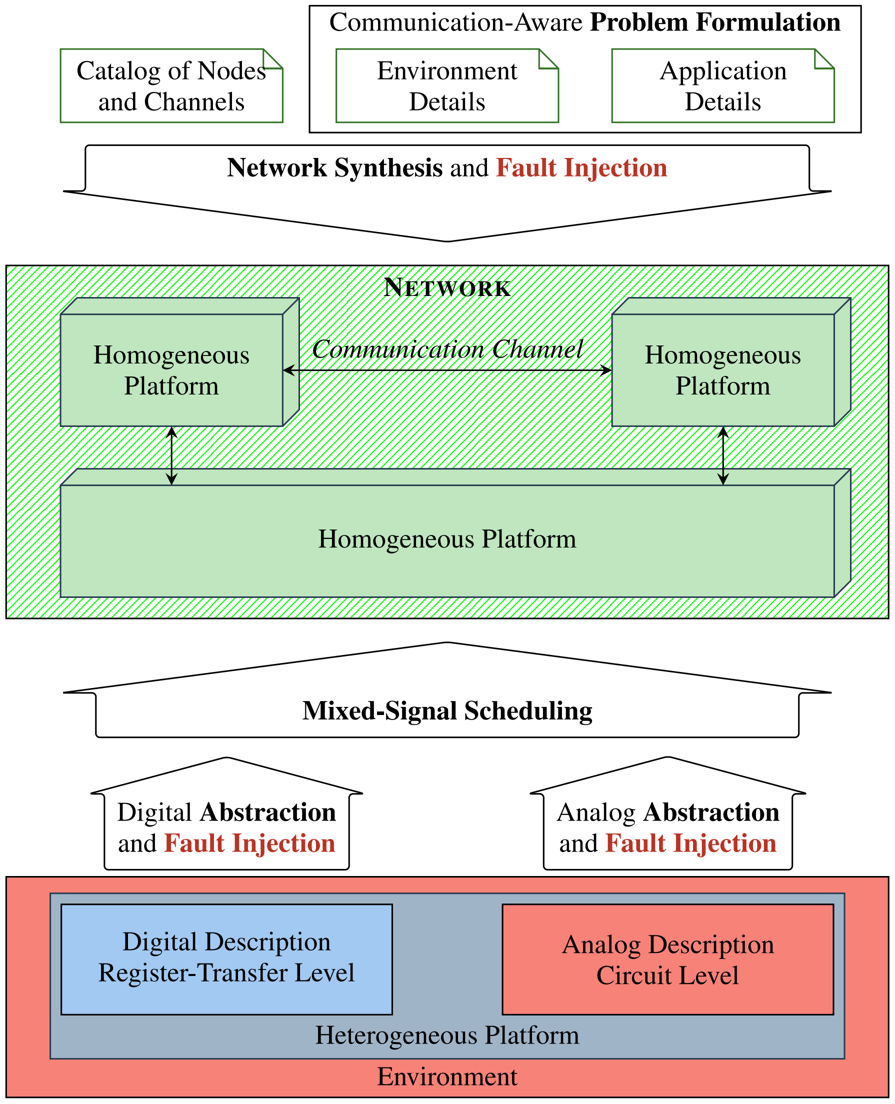

Communication-Aware Problem Formulation
Catalog of Nodes and Channels
Environment Details
Application Details
Network Synthesis and Fault Injection
NETWORK
Homogeneous Platform
Communication Channel
Homogeneous Platform
Homogeneous Platform
Mixed-Signal Scheduling
Digital Abstraction and Fault Injection
Analog Abstraction and Fault Injection
Digital Description Register-Transfer Level
Analog Description Circuit Level
Heterogeneous Platform
Environment

Fig. 1.1: Overview of the unified flow of the thesis.

The *first flow* comes from below and concerns the process of "abstraction" as a mean to unify different simulation domains in a unique homogeneous description.

The starting point of this flow is a heterogeneous platform and its surrounding environment. As the name suggests, this platform is a description containing different components belonging to digital and analog domains. The environment instead, is completely described in the continuous domain and thus considered as an analog description for the purpose of the abstraction. The first step performs the abstraction and fault injection on both digital and analog parts. Then, a second step recombines the abstracted descriptions by creating a mixed-signal scheduler around them. The result is a homogeneous platform written in a high-level language, which for the purpose of this work is C++.

From above the *second flow* starts from the description of a distributed application, the details about the environment, and a list of available channels and nodes. This information is used to perform design space exploration of the network architectures and topologies to find the optimal solutions. The result is a network infrastructure which maps the applications and their flows of communication in specific nodes and channels. Furthermore, each channel is equipped with a user-defined protocol layer which allows injecting faults into transiting packets.
Based on the structure of Figure 1.1, the thesis is organized as follows:

- Chapter 2 presents two techniques for the manipulation of analog components, one technique is for the translation and the other for the abstraction.
- Chapter 3 concerns the entire bottom flow, which is the process of abstraction from the heterogeneous description to the homogeneous one. This chapter is built on top of the abstraction methodology presented in the previous chapter and introduces several extensions. It explains how the abstraction deals with multi-disciplinary analog components and the interfacing between analog and digital processes. It also provides two valuable extensions to the abstraction methodology, to cover transistor-level descriptions, and to simplify the abstraction process for increasing its scalability.
- Chapter 4 concerns the top flow, the process of synthesis for networked cyber-physical systems to automatically generate optimal network infrastructures.

These first three chapters provide the basic concepts that allow us to have a holistic approach to functional safety. The following chapters extend the approaches proposed in the first three to perform a functional safety assessment in each domain of a cyber-physical system:

- Chapter 5 describes in details how the faults are injected in each domain of a networked cyber-physical system.
- Chapter 6 starts by evaluating the simulation efficiency of the three proposed injection techniques. Then, it introduces a simulation scenario which brings together all the techniques in a holistic case study for the diagnostic coverage evaluation.
- Chapter 7 concludes and describes future work.

2

Analog translation and abstraction

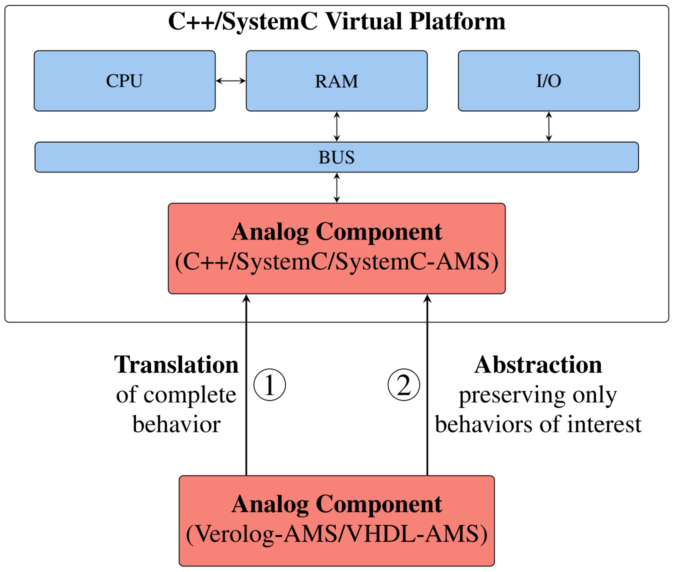


Fig. 2.1: Proposed methodology for simultaneous simulation of analog components with embedded SW and digital HW in virtual platforms.

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 [2]. 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 [3].

The growing importance of the analog domain *w.r.t.* traditional embedded systems has not been compensated by a renewal of the design flows [4]. Embedded SW, digital HW, and analog components follow different design flows, targeting custom technologies and techniques that cannot reconcile extremely heterogeneous aspects [5]. As a consequence, no existing framework or language can handle all aspects of a smart system simultaneously [6].

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 [7, 8, 9]. 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 [10, 11, 12, 13]. Extending the support also to analog components requires the construction of co-simulation frameworks, at the price of an increase of simulation time, that profoundly impacts on time-to-market [14, 5].

In this scenario, this work proposes to enhance the design of smart systems through the *homogeneous 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 2.1, 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 contributions presented in this chapter are:

- the definition of a *translation algorithm* that converts the overall starting description to SystemC-AMS. This flow generalizes the methodology in [14]. 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 subset of input/output relations of interest, to achieve faster simulation. This algorithm makes use of symbolic analysis at generation time, to further remove simulation complexity;
- 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.

This chapter is organized as follows: Section 2.1 provides the necessary background and definitions. Section 2.2 presents the overall approach, that is then detailed in Sections 2.3 and 2.4. Then, both the methodologies are applied to experimental case studies in Section 2.5. Section 2.6 draws our conclusions.

2.1 State of the art and definitions

This section provides all the necessary background, and introduces formalisms that is used throughout this and the following chapters.

2.1.1 AMS extensions of hardware description languages

Verilog-AMS and VHDL-AMS present the same modeling concepts, and their differences are mostly syntactic [11]. Even if the following sections adopt the Verilog-AMS syntax, all considerations are applicable to VHDL-AMS as well.

2.1.1.a 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) [10]. 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).

2.1.1.b 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. The circuit topology can be inferred by analyzing the access functions used inside contribution statements. An access function used between two nodes implicitly defines a branch between them.

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 [15]. This makes AMS simulation very accurate but slow, thus not allowing an effective simulation of mixed-signal systems [16].

2.1.2 SystemC-AMS

SystemC-AMS extends SystemC with constructs for modeling analog and mixed-signal systems [12]. 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 [17] and with its AMS extension [18], 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.

Table 2.1: Taxonomy of analog hardware description levels [1].

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

2.1.3 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 2.1. 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 [1].

Even if the design of analog models is typically top-down [11], recent work proposed bottom-up flows in order to address non-linearities as well as speeding up simulation of analog circuits. In [19] 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. [20] extends the previous work by executing an on-the-fly reachability analysis to select only a subset of the linearized models. Simulation of non-linear analog circuits is also addressed in [21] by applying a state-space exploration technique. Continuous-time models described as a SPICE net-list 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 [22] (both linear [23] and nonlinear circuits [24, 25]) and to reduce complexity of large-scale dynamical models [26] and of multi-physical analog models [27]. However, none of these techniques allow to translate already designed analog models into C++-based languages that may be easily integrated within virtual platforms.

2.1.4 Modeling styles at the base of analog translation

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.

2.1.5 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 the reference node (*i.e.*, *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 an 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:

$$\mathtt{P}_i(b_i) = \left(\sum_{k=1}^{l} C_k \mathtt{P}_k(b_k) \right) + C_{l+1} \tag{2.1}$$

$$\mathtt{P}_i(b_i) = C_i * \mathtt{A}\Big(\mathtt{P}_j(b_j)\Big) \tag{2.2}$$

  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++ (`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).

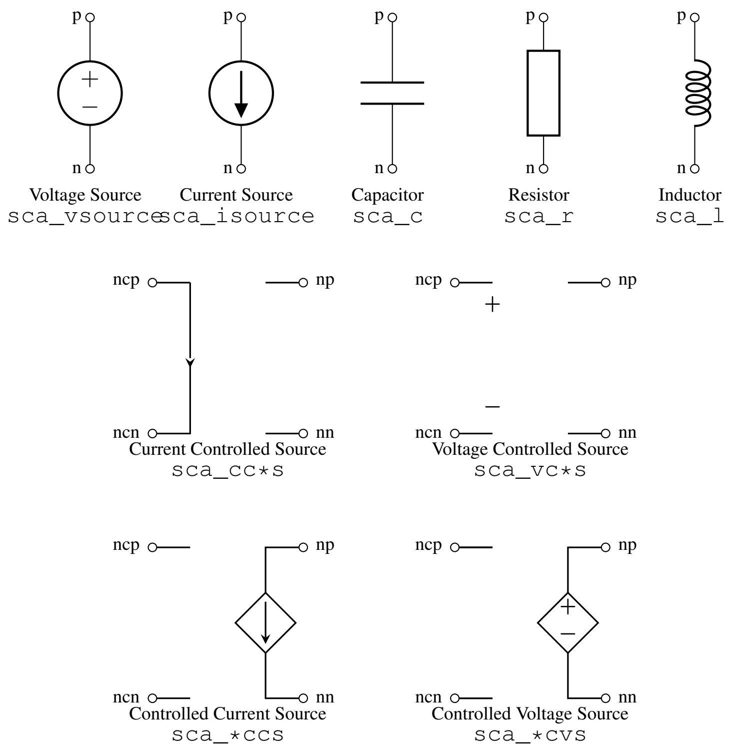


Fig. 2.2: SystemC-AMS ELN terminology and main primitives adopted in this work.

2.1.6 Electrical linear network terminology

Figure 2.2 details the main ELN primitives adopted in this work, as defined by the SystemC-AMS standard [12]. In general, basic ELN modules have a standard interface made up of a positive terminal (`p` port) and a negative terminal (`n` port) for each contributing circuit node. This is the case of basic passive components (*e.g.*, voltage/current sources, resistors, capacitors, and inductors). When an ELN module is controlled by any circuit node, the interface has a source side (*i.e.*, the input of the primitive module, with a positive terminal `ncp` and a negative terminal `ncn`) and a controlled side (*i.e.*, the result of the ELN module, with a positive terminal `np` and a negative terminal `nn`).

2.1.7 Guiding Example

Listing 2.1 shows a synthetic guiding example used throughout this chapter. Its representation in terms of a $\mathcal{S}^{\text{v}} =< \mathcal{N}_e^{\text{v}}, \mathcal{R}^{\text{v}} >$ tuple is as follows:

- electrical nodes are:

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

Listing 2.1: Verilog-AMS code of the guiding example.

```
module example(out, in1, in2, in3);
    output out;
    input in1, in2, in3;
    electrical in1, in2, in3, out;
    parameter real R1 = 1e03;
    parameter real C1 = 100e-09;
    analog begin
        V(in) <+ V(in1) + idt(V(in2) + V(in3)) + 5;
        I(in, out) <+ V(in, out) / R1;
        I(out) <+ ddt(V(out)) * C1;
    end
endmodule
```

- 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:

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

2.2 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 2.1). 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 2.1). This preserves all the physical quantities defined on internal nodes of the system, 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 2.1, 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 subset of values of interest of the system (right-hand side of Figure 2.1): 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 considered as inputs/outputs for the analog device. As

Table 2.2: Models supported by the proposed approaches.

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

a result, the abstraction flow produces a model that is *interface-level equivalent* to the original design and moves an analog model from *circuit* to *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 2.2. 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 [28], 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.

2.3 Translation methodology

The translation implements the flow on the left-hand side of Figure 2.1, and it is represented by a function $\tau(\mathcal{S}^{\mathrm{v}}) = \mathcal{S}^{\mathrm{s}}$, that given a Verilog-AMS implementation $\mathcal{S}^{\mathrm{v}} =< \mathcal{N}_e^{\mathrm{v}}, \mathcal{R}^{\mathrm{v}} >$ returns its node-level equivalent SystemC-AMS implementation $\mathcal{S}^{\mathrm{s}} =< \mathcal{N}_e^{\mathrm{s}}, \mathcal{R}^{\mathrm{s}} >$.

Figure 2.3 gives an overview of the translation procedure. An analog description can be considered as a mean to represent a system of AMS equations composing the circuit. As such, the idea guiding the algorithm is to reproduce the exact set of

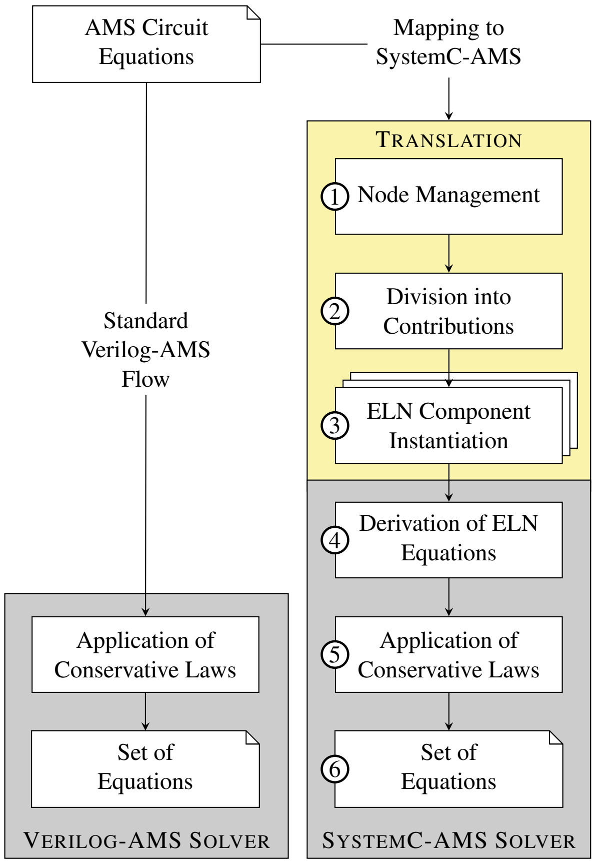

AMS Circuit Equations
Mapping to SystemC-AMS
TRANSLATION
1 Node Management
2 Division into Contributions
3 ELN Component Instantiation
Standard Verilog-AMS Flow
4 Derivation of ELN Equations
Application of Conservative Laws
5 Application of Conservative Laws
Set of Equations
6 Set of Equations
VERILOG-AMS SOLVER
SYSTEMC-AMS SOLVER

Fig. 2.3: Translation flow for analog component descriptions.

equations expressed by the analog description (left-hand side of Figure 2.3) through the instantiation of SystemC-AMS ELN primitives (right-hand side of Figure 2.3).

The translation flow works as follows. Analog nodes are mapped to SystemC-AMS nodes (step ①). Then, every contribution statement is analyzed in order to isolate its basic contributions (step ②). Each contribution is mapped to ideal ELN primitives, where the equations associated with the ELN primitives are the same as the original contribution (step 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 (step ④), that also takes care of applying conservation laws (step ⑤). The resulting equations system will 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 2.2, support can be extended also to non-linear models with some preliminary maneuver. Both strategies must be performed at code generation time, as such, they would not impact simulation performance.

Piecewise-linear models may be supported by applying translation to each linear region individually, as proposed by [19]. 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.*, [19]. These produced piecewise-linear models that can be treated as above.

2.3.1 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. 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 2.1.4). At the same time, the generated code is *conservative*, as ELN primitives predicate over physical quantities of conservative nodes in electrical circuits. As a consequence, they abide by energy conservation laws (*i.e.*, Kirchhoff's laws). This kind of descriptions represents a novel modeling formalism in SystemC-AMS, called Analog Behavioral Modeling (ABM) [14].

The characteristics of ABM models do not fit in any of the SystemC-AMS modeling formalisms [14]. However, they are supported by other AMS HDLs and widely used for the design of components such as MEMS and analog circuitry [29, 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 [12], 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.

2.3.2 Circuit node management

Step $\textcircled{1}$ of Figure 2.3 implements the function $\nu(\mathcal{N}_e^{\mathrm{v}}) = \mathcal{N}_e^{\mathrm{s}}$, 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, then n_i^s must be 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 Listing 2.1, the set of SystemC-AMS nodes is $\mathcal{N}_e^{\mathrm{s}} = \{gnd^{\mathrm{s}}, in^{\mathrm{s}}, in1^{\mathrm{s}}, in2^{\mathrm{s}}, in3^{\mathrm{s}}, out^{\mathrm{s}}\}$, 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`.

2.3.3 Division into contributions

Step ② analyses all the contribution statements and reduces the set of generic relations described by the starting analog implementation into a set of relations expressed in the patterns (2.1) and (2.2) (Section 2.1.5).

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 (2.1), ϵ_2 for a relation expressed in pattern (2.2), and ϵ for a generic expression other than a constant, or an access function.

2.3.3.a Rule 1 – isolating differential contributions

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

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 (2.1). Then, a new contribution in the Form (2.2) is added, to explicit the equivalence between $\mathtt{V}(b_z)$ and the term ϵ_2.

2.3.3.b Rule 2 – managing arguments of differential operators

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

Algorithm 1: Normalization algorithm for the translation procedure.

```
Input  : Initial System (from original specification).
Output: Final Normalized System.
1  $\mathcal{S}' = \texttt{Normalization}(\mathcal{S})$
2      $S' \leftarrow S$
3      foreach $r$ in $\mathcal{R}'$ do
4          if $r \in \epsilon + \epsilon_2$ then
5              $(c_1, c_2) \leftarrow Rule_1(r)$
6              $\mathcal{R}' \leftarrow \mathcal{R}' \setminus \{r\} \cup \{c_1, c_2\}$
7      foreach $r$ in $\mathcal{R}'$ do
8          if $r \in C_0 * \mathtt{A}(\epsilon)$ then
9              $(c_1, c_2) \leftarrow Rule_2(r)$
10             $\mathcal{R}' \leftarrow \mathcal{R}' \setminus \{r\} \cup \{c_1, c_2\}$
11     if $\mathcal{S}' \neq \mathcal{S}$ then
12         $\mathcal{S}' \leftarrow \texttt{Normalization}(\mathcal{S}')$
13     return $\mathcal{S}'$
```

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 (2.2). 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 (2.1) and (2.2).

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), then 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 preserves 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 Listing 2.1. Given the system $\mathcal{S}^{\mathrm{v}} =< \mathcal{N}_e^{\mathrm{v}}, \mathcal{R}^{\mathrm{v}} >$, line 2 creates a new system $\mathcal{S}' =< \mathcal{N}'_e, \mathcal{R}' >$ where $\mathcal{N}'_e = \mathcal{N}^{\mathtt{V}}_e = \{gnd, in, in1, in2, in3, out\}$,

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

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):

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

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

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

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. △

2.3.4 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.

2.3.4.a Type (2.1) contributions

This rule applies to all contribution statements of pattern (2.1):

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

Note that, since we are dealing with linear systems, it is sufficient to take care of the addition of physical values. Figure 2.4 exemplifies the instantiation for the relation:

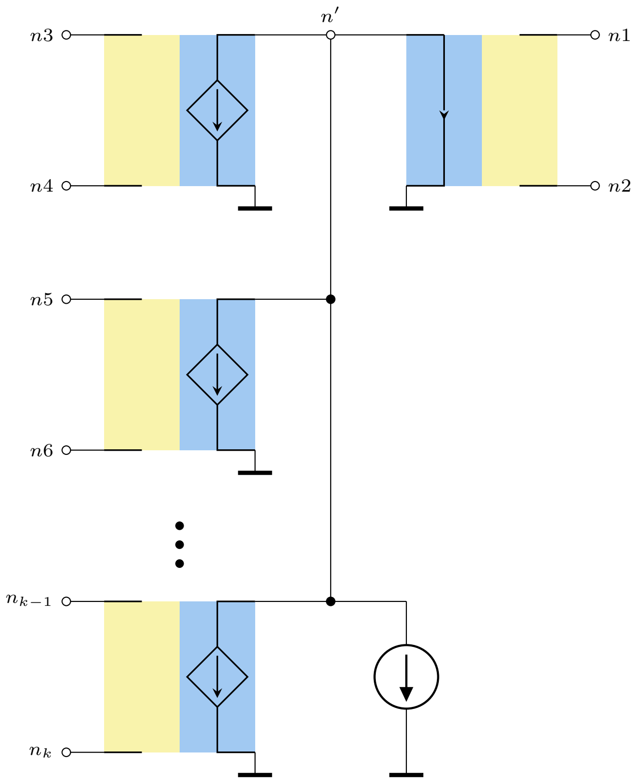


Fig. 2.4: Topological pattern for Type (2.1) contributions.

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

The sum is implemented as parallel composition of controlled current sources (left-hand side of Figure 2.4). 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 a voltage access function (*i.e.*, `V()`), then the instantiated component is a voltage-controlled current source (*i.e.*, `sca_vccs`). If else P_k is a current access function (*i.e.*, `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.

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

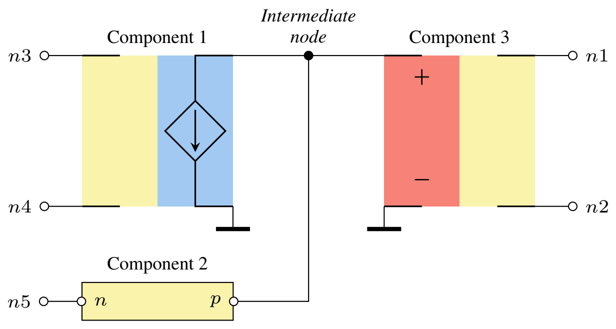


Fig. 2.5: Topological pattern for contributions in Form (2.2).

Let us consider the topology instantiated by the described algorithm from a contribution of type (2.1). 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.

2.3.4.b Type (2.2) 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_c` and `sca_l` respectively. 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 2.5.

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 2.5) 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

Table 2.3: 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`

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:

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

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

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

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 2.3. For each case, the table shows the corresponding SystemC-AMS primitives.

Given a contribution of Type (2.2), the set of equations defined by the topology instantiated as described is equivalent to the original contribution. Let us consider a contribution of Type (2.2), 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 (2.2) 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 2.6 depicts the topology of components instantiated by applying the algorithm to the Verilog-AMS model in Listing 2.1. For the sake of readability, I have omitted the $1G\Omega$ resistors between each node and the ground node. Nodes $in1$, $in2$, $in3$, in and out, are the ones explicitly specified in the original model. Nodes $n1$ and $n2$ are the ones inserted during node management (Section 2.3.2). The other "unnamed" nodes are inserted during ELN component instantiation to connect basic blocks of the topology (Section 2.3.4). △

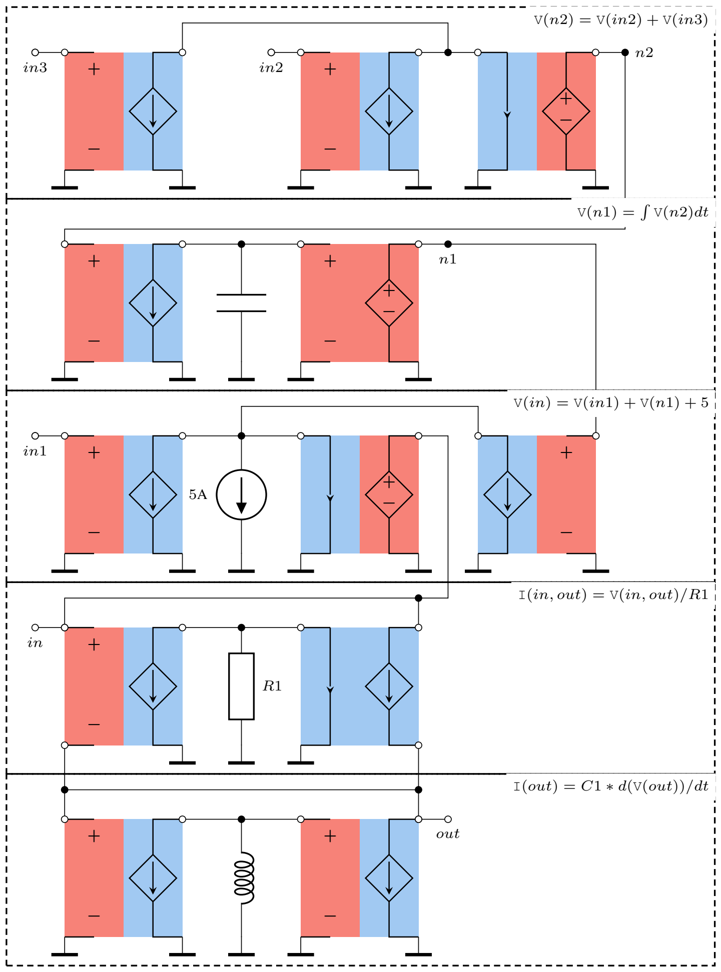
2.3.5 Complexity

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. The complexity for this step is $O(|\mathcal{N}_e|)$.
- *Step* ②: the application of Rules 1 and 2 is constant. 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.*, $O(|\mathcal{N}_e|^2)$). 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 sum 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 $O(|\mathcal{N}_e|^2)$. The complexity of this step is $O(|\mathcal{N}_e|^4)$.

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(n2) = V(in2) + V(in3)
in3
in2
n2
V(n1) = ∫ V(n2)dt
n1
V(in) = V(in1) + V(n1) + 5
in1
5A
I(in, out) = V(in, out)/R1
in
R1
I(out) = C1 ∗ d(V(out))/dt
out

Fig. 2.6: Resulting topology obtained by the translation of the guiding example in Listing 2.1.

2.4 Abstraction methodology

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

Figure 2.7 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.

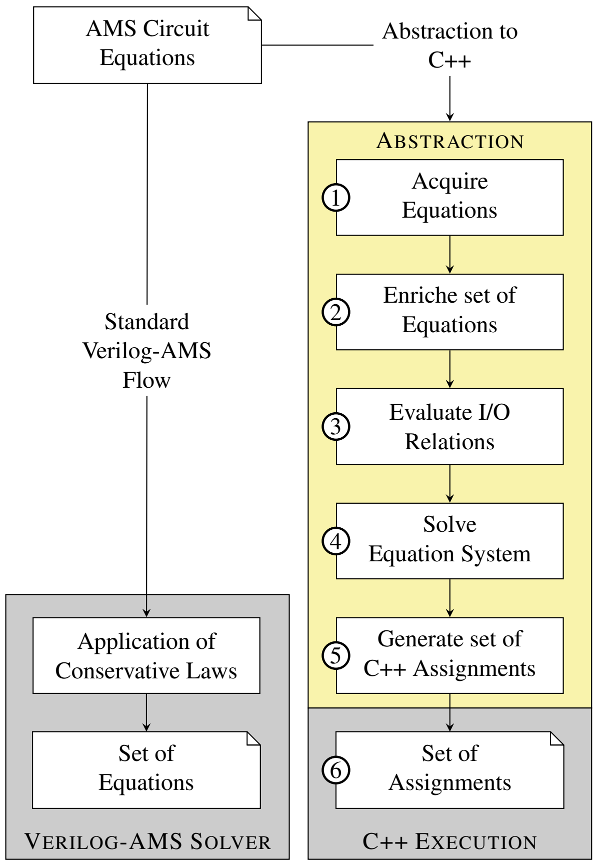


Fig. 2.7: Abstraction flow for analog component descriptions.

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 subset describing the relation between the specified value of interest and the inputs of the model (③). The subset 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).

2.4.1 Circuit equations acquisition

Step ① parses all relations in $\mathcal{R}^{\text{v}}$ 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 $r_i \in \mathcal{R}^{\mathrm{v}}$, ϵ_i identifies the expression on its right-hand side.

2.4.1.a LHS Rule 1

Given any branch $b_{i,j}$, if $\mathcal{R}^{\mathrm{v}}$ 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:

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

2.4.1.b 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.

2.4.1.c 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 $\{n_i\} \cup N_s \cup \{n_j\}$.

2.4.1.d 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:

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

2.4.1.e 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 Listing 2.1 as its formalization $\mathcal{S}^{\mathrm{v}} =< \mathcal{N}_e^{\mathrm{v}}, \mathcal{R}^{\mathrm{v}} >$, where:

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

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

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

△

2.4.2 Equation system enrichment

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

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

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

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

Table 2.4 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.

Table 2.4: Equations gathered (A, B and Z) and generated (from C to H) by the abstraction procedure.

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

The construction of circuit topology insists only on the subset of structural equations $\mathcal{R}^{\mathrm{c}}_{\mathtt{str}}$, due to their adherence to a physical description. *Kirchhoff's conservation laws* [31] allow to derive two new sets of equations: $\mathcal{R}^{\mathrm{c}}_{\mathtt{kcl}}$ and $\mathcal{R}^{\mathrm{c}}_{\mathtt{kvl}}$. $\mathcal{R}^{\mathrm{c}}_{\mathtt{kcl}}$ contains those equations derived from $\mathcal{R}^{\mathrm{c}}$ through the application of Kirchhoff's Current Law (KCL), and it contains $|\mathcal{N}^{\mathrm{v}}_{e}|-1$ independent equations. On the other hand, $\mathcal{R}^{\mathrm{c}}_{\mathtt{kvl}}$ contains $|\mathcal{B}^{\mathrm{v}}_{e}| - |\mathcal{N}^{\mathrm{v}}_{e}| + 1$ 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 2.4. Note that equations describing the same circuit mesh (*i.e.*, E and F) are linearly 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 $\mathcal{R}^{\mathrm{c}}_{\mathtt{str}}$ [32], by interchanging the relation left-hand side term with the right-hand side terms. This introduces a new set of equations $\mathcal{R}^{\mathrm{c}}_{\mathrm{d}}$ linearly dependent from the original one. The size of the new set of equations is $|\mathcal{R}^{\mathrm{c}}_{\mathrm{d}}| = |\mathcal{R}^{\mathrm{c}}_{\mathtt{str}}|$. The application of this step is exemplified by Equations C and D in Table 2.4, which are duals *w.r.t.* Equations A and 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.

2.4.3 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 2.8. The graph is built as follows:

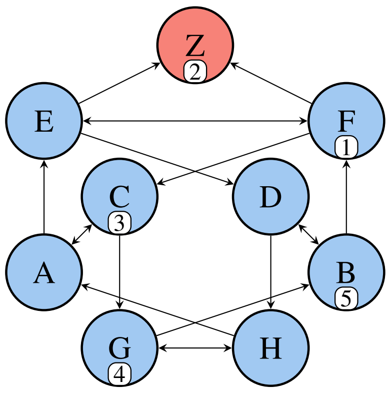


Fig. 2.8: Dependencies graph generated for the guiding example starting from the equations of Table 2.4.

- *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 2.4 and in Figure 2.8 are labeled with letters. Consider node H in Figure 2.8, associated with equation H of Table 2.4, *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 2.8 depicts an example of exploration starting from node F. Numbers represent the order in which the nodes are visited. The resulting set of equations is:

$$\mathcal{R}_{es} = \begin{cases} V\_out = V\_in - V\_in\_out \\ V\_in = V\_in1 + \int(V\_in2)dt + \int(V\_in3)dt + 5 \\ V\_in\_out = I\_in\_out * R1 \\ I\_in\_out = I\_out \\ I\_out = d(V\_out)/dt * C1 \end{cases} \tag{2.3}$$

△

2.4.4 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 2.4, where h is the simulation step of the model:

$$\frac{dV(t)}{dt} \longrightarrow \frac{V(t) - V(t-1)}{h} \tag{2.4}$$

For the integrative operator, a typical example is the quadrature formula in Equation 2.5, where variable V_{acc} is an accumulator incremented by $(V(t) * h)$ at the end of each simulation step:

$$\int V(t)dt \longrightarrow (V(t) * h) + V_{acc} \tag{2.5}$$

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 [28], 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 2.3 is:

$$\mathcal{R}'_{eq} = \begin{cases} V\_in = V\_in1 \\ \qquad\quad + (V\_in2 * h) + V\_in2\_acc \\ \qquad\quad + (V\_in3 * h) + V\_in3\_acc + 5 \\ V\_out = V\_in - V\_in\_out \\ V\_in\_out = I\_in\_out * R1 \\ I\_in\_out = I\_out \\ I\_out = ((V\_out - V\_out\_prev)/h) * C1 \end{cases} \tag{2.6}$$

Listing 2.2: Abstracted C++ code generated for the guiding example in Listing 2.1.

```
void f(double Vin1, double Vin2, double Vin3)
{
    V_in = V_in1 + (V_in2 * h) + V_in2_acc
                 + (V_in3 * h) + V_in3_acc + 5;
    V_out = (C1 * R1 * V_out_prev + V_in * h) / (C1 * R1 + h);
    // Update variables.
    V_out_prev = V_out;
    V_in2_acc += V_in2 * h;
    V_in3_acc += V_in3 * h;
}
```

where `V_in2_acc` and `V_in3_acc` are the accumulator variables introduced by discretizing integrals, while `V_out_prev` is introduced by discretizing the derivative $d(V\_out)/dt$. Listing 2.2 depicts then the corresponding C++ code, obtained through GiNaC. △

2.4.5 Complexity

In the worst-case scenario, the initial set of relations $\mathcal{R}^{\text{v}}$ contains only structural equations. As consequence, the dimension of $\mathcal{R}^{\text{v}}$ 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 $\mathcal{R}^{\text{v}}$. 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, resulting in a complexity of $O(|\mathcal{N}_e|^2)$. 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 $O(|\mathcal{N}_e|^6)$ [28].
- *Step ⑤*: Generating the set of C++ assignments requires at most $O(|\mathcal{N}_e|^2)$.

The overall computational complexity is:

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

Table 2.5: Benchmarks characteristics and generation time.

		Values of Interest		Lines of	Abstraction	Translation
Benchmark	Relations	Input	Output	Code	Time (s)	Time (s)
RC1	2	1	1	17	0.009	0.041
IN2	3	2	2	21	0.012	0.051
PIFilter	4	1	1	21	0.008	0.075
IN3	5	3	2	31	0.015	0.058
Op-Amplifier	6	1	1	31	0.009	0.064
RC5	10	1	1	42	0.014	0.108
RC10	20	1	1	67	0.026	0.197
RC20	40	1	1	117	0.078	0.411
Accelerometer	66	10	8	123	0.020	0.402

2.5 Experimental results

This section shows 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 Analog Systems Translation and absRAction tooL (ASTRAL) tool. ASTRAL relies on the HIFSuite framework [33] for parsing and manipulation of Verilog-AMS models.

2.5.1 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 ideal operational amplifier (*i.e.*, *Op-Amplifier*);
- an accelerometer, modeled using a set of algebraic differential equations expressing behavioral relations over electrical values.

Table 2.5 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.

Table 2.6: 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

2.5.2 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 2.6 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.

2.5.3 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 2.5) is always well below 1 second, for all benchmarks and code versions. To estimate the *simulation performance*, we considered three scenarios for each benchmark:

- Verilog-AMS description, simulated with Questa [15];
- 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 [34]. 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 2.7. 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.

Table 2.7: Execution times for different abstractions simulated both alone and together with the smart system.

	Heterogeneous (Verilog-AMS)		*ASTRAL* automatic translation (SystemC-AMS/ELN)				*ASTRAL* automatic abstraction (C++)			
	Component	Platform	Component		Platform		Component		Platform	
Benchmark	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

2.5.4 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 SMAC Smart System Test Case (S3TC) [5], available as open-source demonstrator for HIFSuite. The structure of the S3TC is depicted in Figure 2.9: 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 S3TC. The SystemC-AMS and C++ versions of the analog components are integrated and simulated within a C++ implementation of the S3TC. 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*.

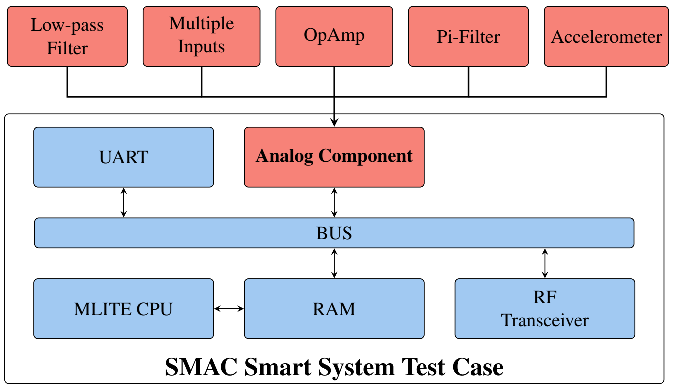


Fig. 2.9: Structure of the SMAC Smart System Test Case virtual platform.

Table 2.7 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 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). The methodology shows improvements in terms of simulation performances in every considered scenario and for all case studies. It allows to effectively and efficiently simulate an entire virtual platform of a smart system, still guaranteeing negligible accuracy losses.

2.6 Concluding remarks

This chapter introduces two methodologies 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.

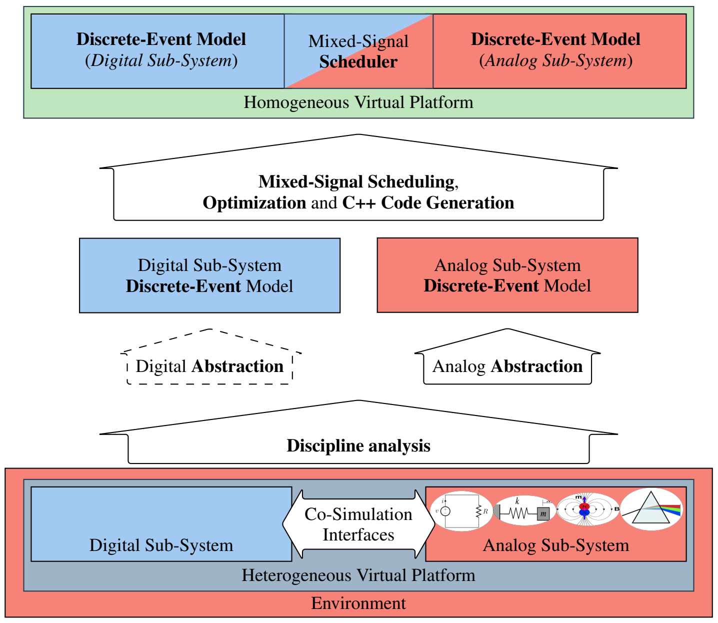
The first methodology concerns the *translation* of an analog circuit to SystemC-AMS, by reproducing the same behavior through the composition of basic building blocks. Each building block is modeled in SystemC-AMS by using ELN primitives of passive components.

The second methodology concerns the *abstraction* of an analog circuit to C++, by moving at generation-time most of the cumbersome operations required to simulated an analog description (*e.g.*, matrix operations, solving a system of equations). It relies on symbolic analysis, a method which in the context of this work is used to solve the algebraic formulas describing the input analog circuit. The result of the symbolic analysis is used to generate the final simulation model.

The effectiveness and correctness of the two proposed techniques have been shown on a number of case studies, as well as on a complete smart system prototype, exhibiting a NRMSE in the order of 10^{-6}. Experimental results highlight the speed ups that can be achieved with the proposed approach, with a maximum achieved speed-up of 711.0x.

3

Analog multi-discipline abstraction and mixed-signal scheduling



Discrete-Event Model (Digital Sub-System)
Mixed-Signal Scheduler
Discrete-Event Model (Analog Sub-System)
Homogeneous Virtual Platform
Mixed-Signal Scheduling, Optimization and C++ Code Generation
Digital Sub-System Discrete-Event Model
Analog Sub-System Discrete-Event Model
Digital Abstraction
Analog Abstraction
Discipline analysis
Digital Sub-System
Co-Simulation Interfaces
Analog Sub-System
Heterogeneous Virtual Platform
Environment

Fig. 3.1: Overview of the abstraction and mixed-signal scheduling approach. All the upward arrows, except for the *digital abstraction*, identify the contributions of this thesis.

The previous chapter introduces the problems related to simulating Analog and Mixed-Signal (AMS) components in the domain of smart systems. It proposes two methodologies for the *translation* and the *abstraction* of analog components, for the purpose of integrating them in smart virtual platforms, and achieve a remarkable speed-up. Until now, we were interested in the analog and specifically the electrical domains. However, most of today's applications for computing systems requires context-aware or the possibility to manipulate the physical environment they are embedded in. This means devices which are able to interact with the physical world: sensing and actuation are usually achieved through the use of components belonging to other domains other than the electrical one (*e.g.*, mechanical, thermodynamic, *etc.*). This chapter focuses on the process of *abstraction* and explores other aspects as well as issues that arise when dealing with such complex multi-disciplinary smart systems.

The concept itself of a *smart system* [35] has been recently introduced to identify energy efficient or autonomous miniaturized devices performing sensing, actuation, communication, and control through computation. This class of systems requires to integrate many different technologies within the same device. Digital hardware and software performing computation must co-exist with analog hardware for wireless communication, as well as integrated sensors and actuators, usually implemented as *Micro-Electro-Mechanical Systems (MEMSs)* [36].

MEMSs are micro-systems made up of components which sizes range from 1 to 100 μm. They provide all the sensing and actuation capabilities required to a smart system while preserving miniaturization. However, their design involves many different physical disciplines. Furthermore, physical processes of different natures (*e.g.*, magnetic and kinematic) may influence each other. For these reasons, such systems are usually modeled as a set of interactions between different descriptions belonging to different disciplines. As such, the simulation of smart devices requires the possibility to manage multi-discipline analog models.

Simulation is crucial when developing novel features. Functionalities of smart devices are usually implemented as software running on top of the hardware platform. In order to validate these applications designers must evaluate the quality of the software running on the device. However, only a simulation able to capture a holistic view of the system [37] may provide enough information. Thus, it is necessary to integrate different simulators for the many disciplines involved, and for the digital hardware performing the computation and communication tasks. Virtual platforms are powerful solutions to perform such a task, however, they mostly focus on digital components. Therefore, continuous-time models need to be integrated through complex and computationally expensive synchronization mechanisms. As shown in the previous chapter, some techniques rely on co-simulation interfaces, which can prove a burden during the simulation. This issue is further exasperated when dealing with many different physical disciplines and natures.

Figure 3.1 shows the structure of the methodology presented in this chapter, which represents the lower part of the big picture shown in Figure 1.1. The approach starts with a platform composed of both digital and analog hardware descriptions and the surrounding environment, which is a continuous-time analog description. Both analog models and especially the environment may belong to different physical disciplines, *e.g.*, magnetic, optic, electrical, kinematic. All the components are translated and then abstracted into discrete-event models. The approach presented in this chapter is based on two state-of-the-art abstraction techniques. The one for the abstraction of hardware models presented in [38], and the one for the abstraction of analog components presented in the previous chapter. However, the mere combination of the two it is not sufficient to abstract complex AMS multi-discipline descriptions. For this reason, the thesis extends the analog abstraction to manage complex *non-linear behaviors*, *continuous-time behavior interacting with discrete-models*, *frequency domain descriptions*, and the *co-existence of multiple disciplines and natures within a single model*.

The work of this thesis aims at overcoming such issues. It proposes an automatic abstraction of multi-discipline models, usually used to represent MEMS components, paired with an automatic abstraction technique for digital hardware models [38]. The two techniques are conjoined by proposing a mixed-signal scheduler generation approach. The models, both digital and analog, are automatically translated into a unifying language. Then, a built-in scheduler is automatically generated by analyzing the relations existing between the models, considering both the digital and analog processes in the system. The proposed approach extends the scheduling and optimization techniques presented in [38] for the digital hardware abstraction. It deals with the interfacing functions which allows the interaction between digital and analog processes. The final result is an executable C++ model able to reproduce with an extremely high-level of accuracy the behavior of the device. The virtual platform can then be used to run, test and validate the SW being developed on top of the multi-discipline, AMS hardware platform.

The methodology explanation is paired with its application to the model of a vibrations motor system, implemented by using MEMS technology. This model is integrated within a hardware platform performing computation and communication. This aims at easing the understanding of our approach while showing its effectiveness on a complex case study. The methodology is then applied to some open-source case studies.

To further extend the contributions of the analog abstraction approach, this chapter provides the basic principles of how to build a *state-dependent small-signal* behavioral-level descriptions written in Verilog-AMS starting from transistor-level netlists. This step allows applying the abstraction to industrial case studies described at transistor-level written in SPICE or SPECTRE. However, such descriptions pose a great challenge in terms of scalability due to their sensible complexity. To overcome this problems, a technique to split behavioral description into sub-systems, which are easier to be dealt with, is provided. This technique is based on the theory of lossless transmission lines. With the increasing interest towards Industry 4.0 and especially smart manufacturing, the last contribution of this chapter concerns the integration of smart platforms inside a factory simulation tool called *Plant Simulation*.
This chapter is organized as follows:

- Section 3.1 gives some background about multi-discipline modeling using Hardware Description Languages (HDLs), discusses the related work available in the literature and summarizes the abstraction technique applied on digital designs;
- Section 3.2 introduces the running example which will be used to exemplify the different issues related to the abstraction of analog components;
- Section 3.3 gives an overview of the proposed methodology;

- Section 3.4 proposes a set of techniques to deal during automatic abstraction with multi-disciplines, complex AMS descriptions;
- Section 3.5 presents the extended scheduling approach developed in this work;
- Section 3.6 presents the methodology use for generating behavioral-level descriptions starting from transistor-level ones;
- Section 3.7 introduces a technique to enhance the scalability of the abstraction methodology, by splitting the system of equations into sub-systems;
- Section 3.8 shows how the abstracted description can be used in the scope of Industry 4.0 and production lines.
- Section 3.9 reports the results obtained by applying the presented methodology to the case study and a set of open-source descriptions;
- Section 3.10 draws the conclusions about the achieved results.

3.1 Background

This section summarizes the main concepts about the system-level modeling and simulation of multi-discipline systems, and the digital automatic abstraction approach combined with the proposed methodology.

3.1.1 Hardware description languages for multi-discipline models

The effort to extend traditional HDLs to model and simulate mixed-system led to make Verilog-AMS and VHDL-AMS the reference languages for analog and mixed-signal design at system-level. Both VHDL-AMS and Verilog-AMS have been used to design components requiring a heterogeneous set of disciplines. Their similarities and differences to accomplish multi-discipline modeling have been deeply analyzed in [11]. They present the same modeling concepts, and the main differences among them are mostly syntactic. This work focuses on Verilog-AMS models, but all the considerations apply to VHDL-AMS as well.

In Verilog-AMS, the behavior of any system is described as a set of relations between physical values on analog nodes and branches. Each system node is associated with two quantities: a potential (*i.e.*, an across value) and a flow (*i.e.*, a through value). Since Verilog-AMS can describe models belonging to different physical domains (including electrical, mechanical, fluid dynamics, and thermodynamics), the characteristics of nodes and quantities are defined in terms of *natures* and *disciplines* to model physical domains [10]. *Discipline* is a type associated with analog nodes, ports, or branches (*e.g.*, electric, magnetic, kinematic, *etc.*). Each nature is a collection of attributes shared by the signals using it. The attributes characterizing a nature are the units, absolute tolerance for convergence, and the name of the function used to access the value of a signal to which the nature is applied. A discipline is associated with one or two natures. In the first case, it is said to be a *signal-flow* discipline, while in the latter it is a *conservative* discipline. A signal-flow discipline specifies either its *flow* or *potential* nature, while conservative disciplines specify both.

Table 3.1: Natures defined by the Verilog-AMS standard.

Nature	Units	Access Function
Voltage	V	`V`
Current	A	`I`
Charge	coul	`Q`
Flux	Wb	`Phi`
Magneto_Motive_Force	A-turns	`MMF`
Temperature	K	`Temp`
Power	W	`Pwr`
Position	m	`Pos`
Velocity	m/s	`Vel`
Acceleration	m/s^2	`Acc`
Impulse	m/s^3	`Imp`
Force	N	`F`
Angle	rads	`Theta`
Angular_Velocity	rads/s	`Omega`
Angular_Acceleration	rads/s^2	`Alpha`
Angular_Force	N-m	`Tau`

Table 3.2: Disciplines defined by the Verilog-AMS standard. Digital Verilog descriptions are considered as models using the *logic discipline*.

Name	Potential	Flow
logic	–	–
electrical	Voltage	Current
voltage	Voltage	–
current	–	Current
magnetic	Magneto_Motive_Force	Flux
thermal	Temperature	Power
kinematic	Position	Force
kinematic_v	Velocity	Force
rotational	Angle	Angular_Force
rotational_omega	Angular_Velocity	Angular_Force

Verilog-AMS standard defines a set of standard natures and disciplines. Table 3.1 summarizes the standard natures, while Table 3.2 lists the standard disciplines and it reports for each of them, the natures used as potential and flow. They specify the types and attributes most used to describe physical values. However, since they are not complete, the language allows designers to define and implement any custom nature or discipline.

In Verilog-AMS the behavior of an analog model is defined as a list of differential equations implemented by using the *branch contribution statement*. A contribution statement is defined by using the *branch contribution operator* (*i.e.*, <+) which is composed by a left-hand side (LHS) and a right-hand side (RHS). The former specifies the physical quantity of a branch to which the value of the latter will be assigned. There are two categories of physical quantities associable to analog nodes: *potential* and *flow*. These can be accessed by means of an *Access Function*, which accepts as arguments two analog nodes and returns the potential difference or the flow between them (*e.g.*, functions `V` and `I` shown in Table 3.1 for the *electrical* discipline). Inside the Verilog-AMS semantics there is a rule that state that: using an access function between two nodes implicitly defines a branch between them. The topology of the circuit can be retrieved by applying such a rule to all the access functions contained inside a Verilog-AMS model.

Verilog-AMS defines an execution semantic to mix the discrete-event model for the discrete processes with numerical techniques necessary to solve continuous time models. Simulation environments usually rely on SPICE-based [13] simulators, making simulation very accurate but slow, and consequently poorly effective [16].

The work in [11] highlights the limitations of Verilog-AMS and VHDL-AMS for system-level simulation. In particular, a guiding example clearly shows the excessively low level of abstraction provided by these descriptions making the simulation of hardware/software systems too slow to allow effective validation. This shows the need for abstraction for analog components.

SystemC-AMS is also worth to be mentioned as the AMS extension of the SystemC language [39]. Due to its system (rather than solely hardware) modeling emphasis, SystemC-AMS differentiates itself from the AMS extensions for Verilog and VHDL described above. It is mostly focused on the system-level representation of a system, providing different modeling formalisms and computational models, allowing specifications at different levels of abstraction. Even if it defines a modeling formalism ad-hoc for electrical descriptions (*i.e.*, SystemC-AMS Electrical Linear Network (ELN)) it still allows to model systems belonging to different physical disciplines. However, they must be explicitly specified by the designer.

SystemC-AMS has been used in the literature to create system-level virtual prototypes of multi-discipline systems [18, 40], thus allowing efficient system-level simulation. However, these models have been re-created in a top-down fashion, manually re-modeling the original implementation of the multi-discipline devices to integrate them into the virtual platform. To enhance this process, [41] proposes an extension to SystemC-AMS allowing to plug in the simulation kernel different models of computation, to ease the modeling of components belonging to a broader set of design domains and disciplines. This further highlights that SystemC is fundamentally a modeling language rather than a design language. It is more suited to develop system-level models for virtual prototyping [40], rather than designing real components. Thus, reusable components to be integrated when designing novel devices are usually available as Verilog-AMS or VHDL-AMS.

3.1.2 Virtual platforms for smart systems

Virtual platforms allow exploring alternative design solutions, and the early validation of the overall system and to develop applications on top of hardware components. They are executable models, allowing to simulate the hardware architecture of a system under development, in order to emulate on top of it the software necessary to implement the desired functionalities.

Most of the currently available virtual platform environments combine C++ and SystemC [7] to achieve fast simulation, while preserving the behavior of the hardware models to emulate. While C++ native constructs and types allow fast execution, SystemC eases integration and reuse of existing IPs. Note that the choice of a single language supporting the overall simulation is crucial, as it avoids the overhead induced by co-simulation, as a result of frequent synchronization between different simulators [5]. However, a unique simulator dealing with the plurality of disciplines and domains involved in the considered kind of systems is not available yet. Thus, a multitude of different simulation technologies has to be employed and connected in order to simulate a virtual platform for such systems. This work aims at proposing a set of solutions to overcome this problem, and the abstraction methodology presented in the previous chapter is the first step in this direction.

3.1.3 Automatic abstraction of digital models

The work presented in [38] presents a set of transformations for HDL descriptions. It allows generating abstracted, highly efficient C++ virtual prototypes of the original hardware descriptions. The main idea is to raise the abstraction level of the component, by moving from Register-Transfer Level (RTL) to Transaction-Level Modeling (TLM). Meanwhile, optimizations are performed to speed up the component simulation further. The methodology relies on three steps.

The first step is the *data-types abstraction*, where the hardware-specific data-types (*e.g.*, logic, logic vectors, *etc.*) are mapped onto C++ native data types. This requires to retrieve data declarations, convert the declared types into the best suited C/C++ types, and finally to reproduce the exact semantics of the operations by using operations over the chosen native data-types. The procedure loses the information related to multivalued logic data-types, but it preserves the functional equivalence *w.r.t.* the original hardware description.

The second step is the *scheduling generation*. A HDL simulation usually requires an execution kernel (or scheduler) governing and managing the sequence of events driving the simulation. An HDL scheduler advances the execution according to the relations defined between processes. This step automatically resolves and generates a scheduler that is embedded within the model to pilot its execution, *i.e.*, to execute the digital processes involved in the description in a correct and highly optimized manner.

The final step is the *code generation*, where the processes described in the HDL descriptions are automatically translated in C++. Then, they are paired with the scheduler generated in the previous step. Finally, the generated code is highly optimized to enhance simulation speed.

The main optimization of this procedure is the scheduling generation. It allows to statically solve all the dependencies among processes while generating the code used for simulation. As such, it moves part of the simulation complexity from runtime to generation time. This process relies on a *Dependencies Graph*: a directed graph specifying all the dependency relations between processes in the model. The graph $G = (V, E)$ is composed as follows:

- V is the set of vertexes of the graph. Each vertex of the graph corresponds to one process of the component. The graph is enriched by a flag `synch` used to state whether the process is synchronous or asynchronous.
- $E \subset (V \times V)$ is the set of edges between vertexes. Each edge $e = (v, v') \in E$ exists if process v' reads a signal written by process v and v'.`sync` if false.

The relations expressed by the graph are sufficient to generate the scheduling procedure for the abstracted description automatically. The dependencies graph presented in [38] is extended in this work. What summarized above is the information necessary to understand the contributions of this thesis. For a more detailed explanation of the graph, we refer the reader to [38].

3.2 Running example

To better explain the different steps of the proposed methodology, the remainder of the chapter will exploit the case study shown in Figure 3.2. It is an AMS model entirely written in Verilog-AMS by assembling and customizing some of the didactic examples used by Ken Kundert in his iconic book on Verilog-AMS design [42] and from the examples released with the Verilog-AMS [10] standard. A simple digital

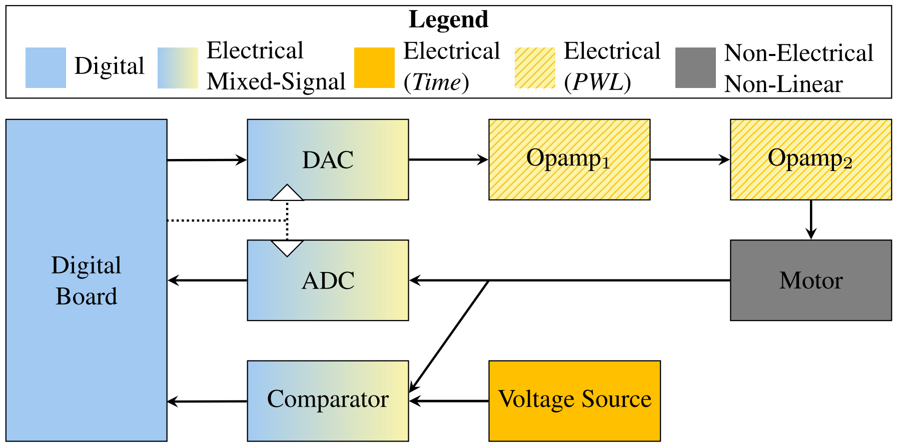


Fig. 3.2: Structure and classification of the components in the vibrations motor system.

hardware platform is controlling the system's AMS components. The example model has been "synthetically" built to exemplify all the issues that are managed in this chapter. This guiding example uses a set of diverse disciplines, linear, piecewise linear and non-linear dynamics, as well as digital-to-analog and analog-to-digital both synchronous and asynchronous communication. Follows a detailed description of each component of the case study.

Digital board

The *digital board* is composed of three processes: two synchronous to a periodic clock signal, and one asynchronous. The first synchronous process generates stimuli and provides them to the Digital-to-Analog Converter (DAC). The second synchronous process reads the values provided by the Analog-to-Digital Converter (ADC) and promptly changes the input stimuli in order to keep the Motor in safe operating conditions. The asynchronous process reacts to the event generated by the *comparator* component whenever the angular velocity of the actuator crosses a fixed safety threshold.

Digital to analog converter

A DAC component converts the digital signals generated by the digital board into voltage values used as input for a series of two ideal Operational Amplifiers (OpAmps). The developed model has a variable resolution, for this test we set it to of 8 bits. The model is linear and based on *logic and electrical* disciplines. The code used in this work is shown in Listing A.4

Ideal operational amplifier

Each ideal OpAmp is an *electrical* model instantiated by using different parameters to specify the minimum and maximum power supply voltages. The amplified output voltage is constrained by thresholds, as such, the OpAmps is described by *piecewise-linear* electrical equations, implemented using the *logic discipline*. The code used in this work is shown in Listing A.2

Motor

The *Motor* component models a mechanical arm rotating an eccentric rotating mass to generate the vibration. The rotation is controlled by the voltage value provided by the DAC, after being amplified by the two OpAmps. It is partially described by using the *rotational omega* discipline, other than the electrical discipline. The mechanical system provides as outputs the shaft position as well as the absorbed potential. The code used in this work is shown in Listing A.3.

Voltage source

The *Voltage Source* component generates a constant voltage used as reference value by the comparator. This component is purely described with the *electrical* discipline.

Comparator

The *Comparator* component uses the value provided by the motor, and compares it with the value generated by the reference source. Whenever the value provided by the rotating mass goes higher than the reference value, the component generates an asynchronous signal to warn the digital board. The comparator is represented by a linear model based on *electrical and logic disciplines*. The code of the comparator is gown in Listing A.6

Analog to digital converter

The *ADC* component converts the voltage representing the angular velocity of the rotating mass into a digital value for the board. It is represented by a linear model based on *electrical and logic disciplines*. The code used in this work is shown in Listing A.1.

3.3 Methodology overview

We have seen the general structure of the analog abstraction and mixe-signal scheduling approach at the beginning of this chapter in Figure 3.1. The methodology starts from an *heterogeneous virtual platform* describing the system under design by employing a mixture of different HDLs and their AMS extensions. The model undergoes a *Mixed-Signal Analysis*, which identifies the parts of the design belonging either to the digital or the analog domain. Then, the model is split into two sub-models:

- the *Analog sub-model* is composed by any instance of design unit or module whose architecture contains at least one construct strictly belonging to an analog or AMS extension of an HDL (*i.e.*, Verilog-A, Verilog-AMS, VHDL-AMS).
- the *Digital sub-model* is composed by all the instances of design unit and modules that do not belong to the analog sub-model, *i.e.*, all the design units and modules specified by strictly using only constructs defined by the standard HDLs (*i.e.*, Verilog and VHDL).

The analysis stores the timing information about the synchronization between the two sub-models:

- the *sensitivity lists* of processes in the analog sub-model, originally communicating with signals in the digital sub-model.
- The set of *events* generated by processes in the analog sub-model that influences processes in the digital sub-model.

- The set of signals used to connect analog and digital models.

The two sub-models undergoes two different flows after being identified. The digital sub-model is abstracted by applying the techniques presented in [38] to obtain a functionally equivalent abstract model that relies on transactions rather than register transfer level internal communication, that can be automatically translated into a C++ virtual prototype. On the other hand, the analog sub-model need to be reconciled with the digital part of the platform, in order to achieve fast homogeneous simulation [37]. For this reason, the analog sub-model undergoes analog model abstraction.

This chapter generalizes the analog abstraction methodology presented in Chapter 2 to deal with more physical domains other than the electrical one. This generalized flow starts with *discipline analysis step*, enabling the possibility to manage multi-discipline AMS models, and proposes several extensions to the abstraction methodology. In particular, Section 3.4 presents all the improvements necessary to support AMS (rather than just analog), and multi-discipline (rather than purely electrical) models. Furthermore, the analog abstraction identifies (if present) a set of constructs that impact on the digital sub-model of the system, such as Verilog-A's timed statements.

After abstracting the two sub-models, they are re-composed to create a *Homogeneous Virtual Platform*: a C++ virtual prototype reproducing the behavior of the system at a very abstract level, while preserving its input-output relations. The re-composition is performed through a *Scheduling and Synchronization* step that schedule and synchronize the processes of the two sub-models. This step is based on the scheduling approach presented in [38], and it extends the previous work to exploits the synchronization information obtained during the mixed-signal analysis and the analog model abstraction steps. The generated scheduler is embedded within the system model. Finally, the model is automatically optimized and synthesized into a C++ virtual prototype through automatic code generation.

3.4 Manipulation and abstraction of multi-discipline analog models

The abstraction methodology starts from the (potentially) analog sub-model of the system and generates its C++ executable model. The final result preserves the relations between the input and output identified for the system. Figure 3.3 depicts an overview of the abstraction flow presented in this section.

The initial multi-discipline analog description is analyzed by the *Discipline Analysis* step. This step identifies which disciplines are used in the different parts of the system. The analysis is composed of two steps:

- the *Flattening* step transforms a hierarchical description into a non-hierarchical one while preserving the behavioral equivalence *w.r.t.* the original model.
- the *Splitting* step visits the flattened system, identifies the parts using different disciplines and split the model accordingly. Therefore, it partitions the multi-discipline flattened model into mono-discipline sub-models.

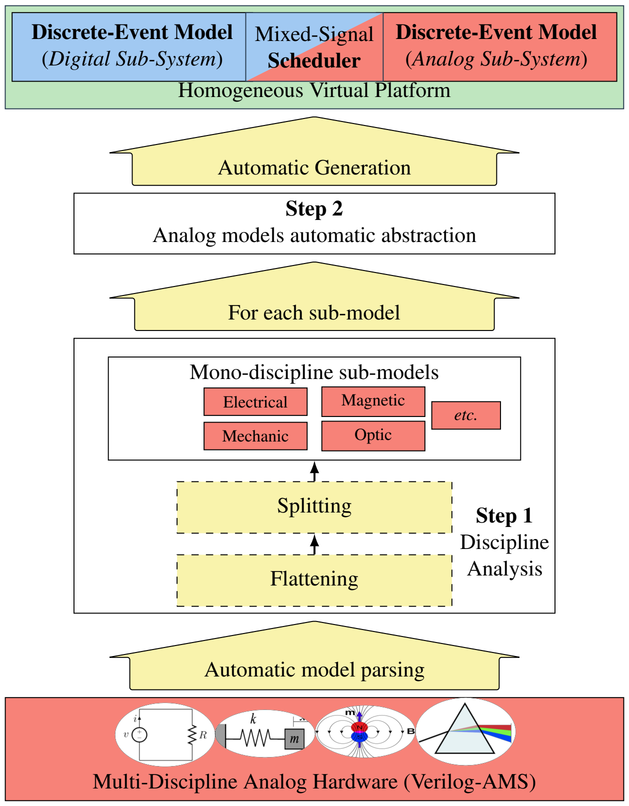


Fig. 3.3: Overview of the abstraction procedure for multi-discipline analog models.

The Discipline analysis step, further discussed in Section 3.4.1, is required because a set of nodes of the same conservative discipline, and connected to each other, must be considered as a unique system [10].

After the discipline analysis, each of the mono-discipline models produced is abstracted in the *sub-model abstraction* step. This step is built on top of the state-of-the-art abstraction methodology for electrical analog models presented in Chapter 2. However, in the following of this section it is extended to deal with models expressed in the frequency or Laplace domains (Section 3.4.2), non-electrical disciplines and natures (Section 3.4.3) and non-linear behaviors (Section 3.4.4).

3.4.1 Disciplines analysis

The discipline analysis starts by flattening the design. The guiding idea of the *Flattening Process* is: given a hierarchical model, analyze the boundaries between its sub-models, and merge them into a single non-hierarchical description. The process building the flattened description is organized in four steps:

1. For each instance of any sub-component of the analog sub-system, the procedure creates a copy of the sub-component and it adds the copy to the flattened description.
2. For each sub-component copied into the flattened description, each of its input and output port (*i.e.*, terminal) is replaced by an intermediate node. The discipline of the new intermediate node will be the same of the original port.
3. For each variable imported into the flattened description must be renamed to avoid name clashes. The variables are imported when creating the copies of the sub-components. A side effect of the flattening copy will be the elimination of some scopes. As such, if different variables in different sub-components have the same name, name clashes are possible. Each variable is renamed by adding as a postfix to its name. The postfix will be created by analyzing the instances of components in the hierarchy where the variable is contained. For instance, if a variable `x` is declared within the sub-module `sub` of the module `top`, the new variable name will be: `x_sub_top`. Finally, to further eliminate the change of having naming conflicts, a randomly generated alphanumeric sequence is added as a postfix to the new name.
4. The internal behavior of each sub-model is modified to be compliant with the previous manipulation. Any reference to internal ports of components is replaced by the corresponding variable introduced by the second step. Any reference to any variable is replaced according to the renaming performed by the third step.

Figure 3.4 depicts the non-hierarchical model resulting by applying the discipline analysis to the vibration motor case study. Boxes with the same color identify areas of the system described using the same discipline and the same modeling style. As the figure shows, the internal behavior of each sub-model has been exposed and

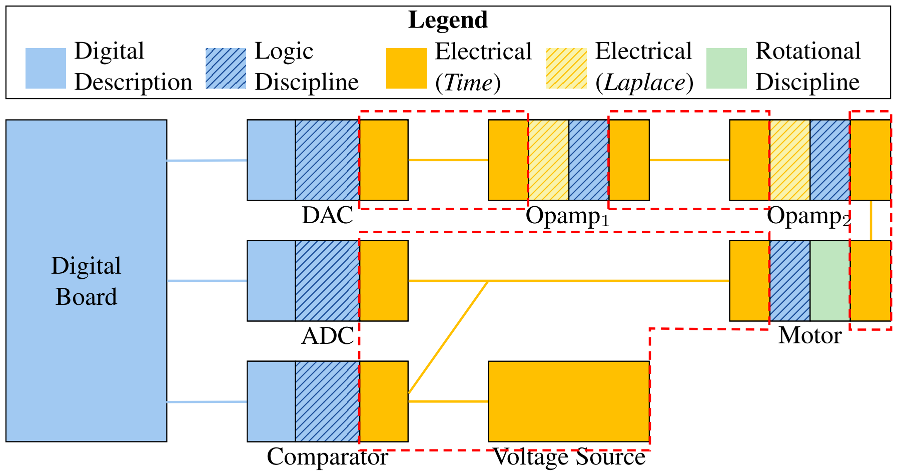


Fig. 3.4: Disciplines analysis for the vibration motor case study.

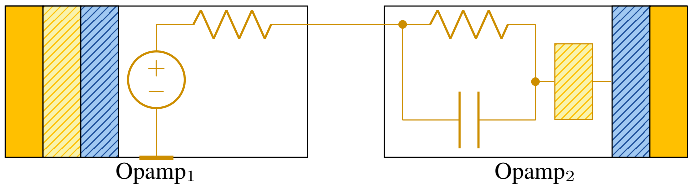


Fig. 3.5: Electrical circuit defined inside different modules, outlined by the discipline analysis step.

partitioned into its different disciplinary components. Dashed red lines shows the sub-models described with the same domain connected by conservative nodes.

Once the first part of the discipline analysis is completed, the flattened description is analyzed using the splitting process. Such step outlines and divides all the areas of the system which, by belonging to the same discipline, need to be considered as a single model. One-by-one, all the identified sub-models are going to be abstracted separately. The abstraction will produce a procedure for each sub-model. Then, the methodology recomposes all the procedures through scheduling (Section 3.5) to produce the final system prototype.

Figure 3.5 shows a portion of the case study defined using the electrical discipline, and identified by the presented analysis. The circuit in figure, contains a voltage source and a resistor inside the OpAmp_1 component, and a capacitor in parallel with a resistor originally defined inside OpAmp_2. Consider the case of the two OpAmps, they are two different instances of the same component. At first glance, it may seem reasonable to abstract the definition of the component and replicate twice the resulting abstracted model. However, this will not preserve the correct behavior of the composed system, since once instantiated they are sharing a net composed by conservative nodes (*i.e.*, the intermediate node interconnecting the two OpAmps in figure).

The voltage source and the resistor inside the OpAmp_1 are influencing the passive components (*i.e.*, resistor and capacitor) inside the OpAmp_2, and vice versa. As a consequence, it is necessary to consider the composition of electrical components depicted in Figure 3.5 as a unique system of equations. It is worth noticing that concerning the components which make use of the electrical discipline, they have been modeled partially using the Laplace-domain. However, since the splitting process focuses only on disciplines, rather than modeling styles, the time- and Laplace-domain parts are grouped within the same sub-model. Therefore, they must be considered as part of the circuit spread over the two OpAmp instances. Occurrences of the issue just discussed can be easily exposed by applying the flattening step, while they can be solved by applying the splitting procedure.

3.4.2 Frequency domain

Linear time-invariant systems are commonly expressed using their transfer function in the frequency domain. This kind of specification allows to represent a linear dif-

Listing 3.1: Manipulation of an electrical sub-model, to remove the Laplace-domain specification.

```
V(out_dac) <+ value_dac;
I(in_opamp_1) <+ (V(in_opamp_1) / rin_opamp_1) +
                 ddt(V(in_opamp_1) * cin_opamp_1);
A_opamp_1 = laplace_nd(V(in_opamp_1) * C1, 1, {1, C2});
```

```
V(out_dac) <+ value_dac;
I(in_opamp_1) <+ (V(in_opamp_1) / rin_opamp_1) +
                 ddt(V(in_opamp_1) * cin_opamp_1);
A_opamp_1 = C1 * V(in_opamp_1) - C2 * ddt(A_opamp_1);
```

ferential equation using an algebraic equation, easier to manage and solve. However, it provides a frequency domain representation that must be solved (*i.e.*, applying the inverse Laplace transform) during time-domain simulation.

To move resolution overhead from simulation- to generation-time, whenever a contribution statement using the Laplace notation is found while parsing the design, its inverse Laplace transform is symbolically computed to retrieve the corresponding time-domain representation. This is done by applying simple manipulations to the Abstract Syntax Tree (AST) representing the transfer function. This will produce a novel AST representing an equivalent time-domain function that is collected together with the other time-domain equations composing the mono-discipline sub-model. As such, all the equations composing the sub-model are expressed in the time-domain and they can be solved altogether.

The operational amplifiers inside the vibration motor case study are partially modeled in the Laplace domain: they undergo the inverse transformation. Listing 3.1 depicts the Verilog-AMS code of the electrical sub-model spread among the digital-to-analog converter and the first operational amplifier (*i.e.*, OpAmp$_1$), obtained after the discipline analysis step. The upper part of the listing shows the original model, where the behavior of the variable `A_opamp_1` (initially called `A`) is specified in the frequency domain, by using the `laplace_nd` function. The `laplace_nd` is an operator which allows implementing a Laplace transfer function in terms of numerators and denominators of the form:

$$H(s) = \frac{n_0, n_1 \cdot s, n_2 \cdot s^2, \ldots, n_m \cdot s^m}{d_0, d_1 \cdot s, d_2 \cdot s^2, \ldots, d_m \cdot s^m}$$

The lower part of Listing 3.1 shows the Verilog-AMS code describing the model after the inverse transformation: the result is specified in the time-domain.

3.4.3 Conservative disciplines and custom disciplines

The set of equations explicitly specified by each contribution statement are stored into a multimap data-structure. For each entry, the key in the map is given by the left-hand side of the contribution statement, while the map's value is the AST generated parsing the right-hand side. However, the set of equations lacks all those implicit equations implied by the explicitly specified ones. The enrichment sub-step takes care of enriching the multimap with the implicit equations of the system.

The first set of equations to be inserted is composed by all the equations obtained by finding the *dual relations of each equation*: for each equation, the left-hand side value is interchanged with all the terms contained on its right hand side. This procedure is applied to any set of equations for both conservative and signal flow descriptions.

Conservative disciplines make necessary to take into account all those equations that are implied by the application of energy conservation laws. Previous approaches were using the Kirchhoff's Voltage and Current Laws, thus, they were applied only to the electrical discipline. However, the Verilog-AMS standard relies on their generalization: Kirchhoff's Potential Law (KPL) and Kirchhoff's Flow Law (KFL) [10].

For each mono-discipline sub-model extracted after the discipline analysis: if the discipline is conservative, the equations stored in the multimap are exploited to gather information about the structure of the system. The procedure identifies the conservative relations between physical quantities. This process is a generalization of what already presented in the literature to manage conservation laws for electrical circuits:

- For each node belonging to the conservative discipline: the procedure identifies which equations are using the node's potential access functions.
- The equations retrieved are used to apply the KFL introducing a new equation in the system to assure that the sum of all flows out of the node is zero.
- For each branch used in the model: the equations using the flow access function on the branch are identified and related to each other in order to identify all the loops in the system.
- For each loop: the KPL is applied to force in the system an equation ensuring that the sum of the branches potentials around the loop is equal to zero.
- All the equations generated by applying the conservation laws are used to enrich the multimap used to assemble the AST of the assignments linking inputs and outputs of the sub-model.

The vibration motor case study presents three different disciplines: *logic*, *electrical*, and *rotational_omega*.

As a consequence the discipline analysis step presented above provides the means for splitting the model of the device into four electrical sub-models, and a rotational_omega sub-model. The electrical sub-models have been managed individually to apply the Current and Voltage laws, thus retrieving the hidden electrical relations. Analogously, the rotational_omega sub-model has been analyzed by applying the generalized KFL and KPL.

3.4.4 Non-linear behavior

Verilog-AMS provides different ways to introduce non-linear behaviors inside an analog description, this work considers the following three:

1. the logic discipline can model piecewise-linear behavior, by using control flow statements (*e.g.*, if-then-else, *etc.*);
2. analog functions that may define any kind of dynamic behavior, usually by using mathematical functions (*e.g.*, polynomial);
3. and contribution statements on conservative nodes can contain non-linear functions on their right-hand side.

This work addresses these three issues separately.

3.4.4.a Control flow statements

The splitting procedure carried on by the discipline analysis step isolates all the portions of the models that are modeled using the Logic discipline. Furthermore, the Verilog-AMS standard forbids to use analog nodes or branches access functions inside a logic block [10]. Therefore, to interface the Logic discipline with conservative or signal-flow discipline it is necessary to introduce real variables in the Verilog-AMS model. Then, these variables are the only one used within a partition of the system expressed using the logic discipline. Thus, the semantics of such partitions become purely procedural. This work exploits such a feature and it reproduces non-linear behaviors introduced by the logic discipline by translating the procedural logic block into an equivalent C++ function.

The code in Listing 3.2 exemplifies the application of translation for logic discipline non-linear behavior. The upper part of the listing shows a portion of a simplified Verilog-AMS implementation of the operational amplifier defined in the guiding example. Variables A and C are used to interface logic and electrical discipline. Then, a conditional statement on A introduces a non-linear behavior. The lower part of the listing shows the corresponding C++ implementation. The function `logic_block_1` implements the behavior specified using the logic discipline in the Verilog-AMS code, while the `opamp` function implements the behavior specified by the Verilog-AMS code.

Listing 3.2: OpAmp component of the vibration motor case study.

```
A = V(in) * gain;
if( A > threshold ) C = threshold;
else C = A;
I(out) <+ C;
```

```
void logic_block_1(double & A, double & C) {
    if ( A > threshold ) C = threshold;
    else C = A;
}
void opamp() {
    A = V_in * gain;
    logic_block_1(A, C);
    V_out = C;
}
```

It is worth noticing that this kind of behavior is typically used to interface different disciplines. The application of the presented strategy must be applied when aggregating the abstracted mono-disciplinary sub-models after the abstraction step.

3.4.4.b User defined analog functions

A non-linear behavior may be introduced by an analog function. This case is managed by producing the equivalent C++ implementation of the analog function. This closely recalls the strategy adopted for the case of non-linear dynamics induced by interfacing logic and conservative disciplines. In this case, however, we must discern two cases:

1. the non-linear is described by using a standard Verilog mathematical function (*e.g.*, trigonometric functions). The behavior is reproduced by using the corresponding function of the standard C++ library. For instance, by mapping the original Verilog function onto a function defined within the `cmath` header of the C++. The methodology is accompanied by a support library containing the implementation of any function defined in the Verilog-AMS standard [10] and does not have a correspondence within the C++ Standard Library.
2. the non-linear behavior is described by a custom function defined by the designer. The abstraction flow produces a C++ function that is the translation of the custom Verilog function. The translation is based on the analysis of the AST of the Verilog custom function, that is then reproduced by using C++ constructs. If the custom function contains calls to standard Verilog functions, these are mapped as described for the case above.

The first case is pretty straightforward, while the second case is more complicated. Let us consider a custom analog function such as the one shown in Listing 3.3. The code shows the application of this strategy to reproduce the behavior of Verilog-AMS analog functions in C++. The upper part of the listing shows the original analog function, while the lower part shows the corresponding C++ implementation. It is also worth noticing the translation of the standard `pow` and `sin` Verilog functions,

Listing 3.3: Example of custom function specified in Verilog-AMS (above), and the corresponding C++ implementation (below).

```
analog function real custom_function_1;
    input a, b, c, x;
    real  a, b, c, x;
    custom_function_1 = a * pow(x,b) + c * sin(x);
endfunction;
```

```
#include <cmath>
double custom_function_1(double a, double b, double c, double x) {
    return a * pow(x,b) + c * sin(x);
}
```

that is automatically performed by using the `pow` and `sin` functions defined within the Standard Library of the C++ language, within the `cmath` header.

3.4.4.c Contribution statements

If a contribution statement defines a non-linear behavior, then it has to be considered as a signal-flow equation. Thus, it is merely parsed and its abstract syntax tree is inserted into the multimap data structure built while parsing the input design.

3.5 Mixed-signal scheduling for system integration

We have seen in Section 3.1 and previously in Chapter 2 that the automatic abstraction of digital and analog designs produces a set of C++ functions, each of which implements the behavior of a specific process involved in the original mixed-signal description. Synchronization and scheduling of these C++ functions are necessary to imitate the behavioral evolution of the initial description correctly.

This section first addresses how to generate an embedded scheduler within the abstracted model automatically, then it presents two cross-domain functions which are critical for handling mixed-signal simulations.

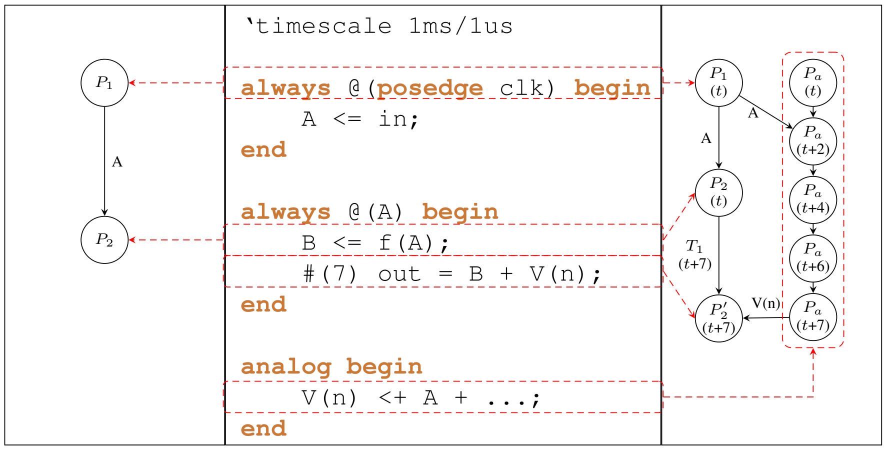
3.5.1 Model temporization

The proposed approach relies on a *dependency graph*, expressing dependencies among processes. It is a direct graph, where nodes represent digital processes of the platform. An edge connects two nodes if the process represented by the target is reading a signal written by the process represented by the source of the edge. The analysis starts from the synchronous processes. Afterward, the graph is built by analyzing all the dependencies of asynchronous processes.

Figure 3.6.a is a simple example of a dependency graph obtained by analyzing the digital processes of the Verilog code in Figure 3.6.b. The only piece of information represented there is that P_2 depends on P_1 through the digital signal A. However, it is worth noting that the source code of P_2 introduces an explicit delay of 7 ms, that is no longer visible in the graph. Thus, it is necessary to introduce timing information in the graph.

The user has to provide information about the signal to be considered as system clock together with its period to perform the timing analysis. This allows to identify all paths in the graph that start with the synchronous process and traverse all the asynchronous ones: the time to execute all the processes in a path must be equal to the clock period.

The second piece of timing information is given by the explicit delays specified in digital processes. They give an exact timing to asynchronous processes that may need to be synchronized with the analog ones. The dependency graph is slightly modified, to take care of this: Processes with explicit delays are split into sub-processes, each



```
`timescale 1ms/1us

always @(posedge clk) begin
    A <= in;
end

always @(A) begin
    B <= f(A);
    #(7) out = B + V(n);
end

analog begin
    V(n) <+ A + ...;
end
```

Digital Depndency Graph (a) | Verilog Source Code (b) | Enriched Depndency Graph (c)

Fig. 3.6: Example of dependency graph enrichment, starting from a Verilog-AMS description.

one representing the part of the process between two delays. The parts are connected by an edge representing the timing relation.

Figure 3.6.c depicts how the graph is enriched with this information. A new node P_2' is introduced representing the piece of process executed after the delay. Then, P_2 and P_2' are connected through an edge labeled with a new event (*i.e.*, T_1), by representing the dependency and the relative timing with respect to the synchronous process.

Timing information allows inserting analog processes in the graph correctly. Their execution frequency f_a is $f_{clock} * k$, where f_{clock} is the clock frequency, and k is an integer constant. Thus, k executions are inserted in parallel into the graph and annotated with their timing. Two kinds of dependencies can occur between digital and analog processes:

- An analog process reads a value written by a digital process. An edge is inserted from the "digital node" to the "analog node" annotated with the next time stamp with respect to the one of the digital node.
- A digital process reads a value written by an analog process. An instance of the analog process is inserted and labeled with the time stamp of the dependent digital process to force analog execution at that particular time stamp.

In Figure 3.6.c the analog process P_a is dependent on the value of A available one delta cycle after the execution of P_1. The value of A is available at the second execution of P_a due to concurrency. Then, P_2' depends on the voltage value of the electrical node n: a new instance of P_a is introduced to execute the analog process at time $t+7$, where t is the last tick of the clock signal occurred. Then, a dependency from P_a at time $t + 7$ and P_2' is inserted in the graph.

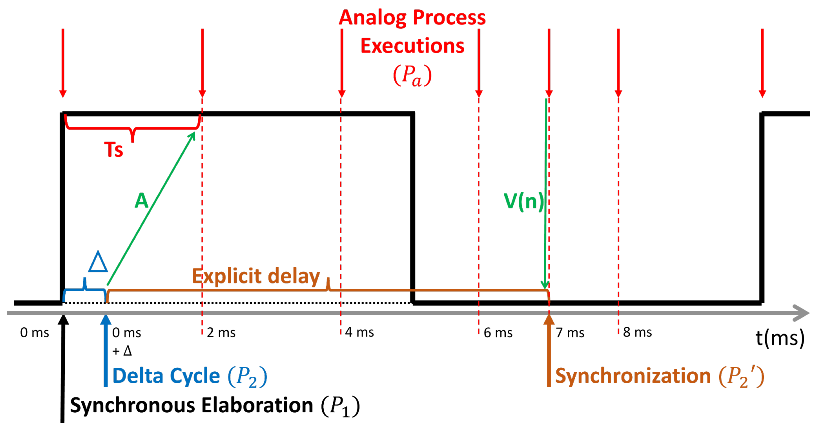

Analog Process Executions (P_a)
Ts
A
V(n)
Δ
Explicit delay
0 ms
0 ms + Δ
2 ms
4 ms
6 ms
7 ms
8 ms
t(ms)
Delta Cycle (P_2)
Synchronization ($P_2{}'$)
Synchronous Elaboration (P_1)

Fig. 3.7: Clock cycle execution of the example in Figure 3.6, applying the proposed scheduling approach.

3.5.2 Temporal decoupling and Synchronization

The enriched graph is used to generate the final scheduling. The set of edges in the graph can be interpreted as a partial order relation. All the processes that are not related by this relation can be decoupled and simulated independently. In Figure 3.6.c the execution of P_2 is independent with respect to P_a at time $t+2, t+4, t+6, t+7$. Hence, the execution of P_2 and P_a in the time range from $t + 2$ to $t + 7$ can be decoupled.

Then, it is necessary to synchronize the global time of the system and execute the processes in the order imposed by the relation, whenever the partial order relation introduced by the dependency graph exists. This happens for instance between P_1 and P_a at time $t+1$ and between P_a at time $t+7$ and P_2'. Figure 3.7 depicts the time evolution of the simulation of one clock cycle of the system exemplified above (Figure 3.6). Arrows indicate timing of events happening in the system. Red arrows represent executions of the analog process P_a. A black arrow depicts the execution of the synchronous process P_1 while a blue arrow indicates the execution of the asynchronous process P_2. Finally, at 7 ms analog and digital processes are synchronized as discussed above.

In this way, temporal decoupling allows to synchronize analog and digital processes only when they influences each other rather than synchronize them at every time step as done by mixed-signal HDLs and SystemC-AMS. Thus, it provides lighter synchronization while preserving the AMS semantics.

3.5.3 Cross-domain analog functions

Special care is required whenever *timing statements* are used to perform time/event-triggered tasks for the purpose of interfacing discrete-time and continuous-time domains. We can find functions which are applied to just one statement (*e.g.*, transition), or to an entire block (*e.g.*, initial_step, cross, timer, above, *etc.*). In either case, these functions generate events at specific instants in time during the simulation, and failing to deal such events can lead to a non-equivalent simulation.

3.5.3.a Transition

The *transition* function is commonly used to interface the digital domain to the analog one. It allows to smoothly drive a continuous time signal with a discrete one avoiding abrupt changes of values which can lead to discontinuities. The transition happens as soon as the expression driving the function changes. Here the issues to take care of are several. First, the simulator needs to start the transitioning process as soon as possible, which implies that the simulation step should be small enough to capture the changing in the driving expression accurately. Second, the driver could change during the execution of the function, in this case the transition needs to be re-scheduled.

Listing 3.4 shows one possible implementation of the transistor function written in C++. A transition function is implemented through a slope from the initial value of the expression to the new value. A designer can customize the slope behavior by specifying an initial delay and a rising/falling time. The steepness of the slope

Listing 3.4: Transition function implementation.

```cpp
inline void Transition::transition(
        double expr, double tdelay, double trise, double tfall) {
    // Re-compute slope and y-intercept if expression has changed.
    if (!dbl_equal(expression, _y1)) {
        _x0 = _system_abstime();    // The current time.
        _x1 = _x0 + tdelay;         // Current time plus the delay.
        _y0 = _output;              // The current y value.
        _y1 = expression;           // The final y value.
        // Based on the direction add the rising or falling time.
        if (dbl_lequal(_y0, _y1) && (trise > 0.0)) {
            _status = Status::RISING;
            _x1 += trise;
        }
        else if (dbl_gequal(_y0, _y1) && (tfall > 0.0)) {
            _status = Status::FALLING;
            _x1 += tfall;
        }
        // Compute slope.
        _m = (_y1 - _y0) / fmax(1E-12, _x1 - _x0 - tdelay);
        // Compute y-intercept.
        _b = _y1 - _m * _x1;
    }
    // Check if the intial delay is elapsed.
    if (dbl_gequal(_system_abstime(), tdelay + _x0)) {
        // Evaluate the output signal.
        _output = _system_abstime() * _m + _b;
        if (((_status == Status::FALLING) && (_output < _y1)) ||
            ((_status == Status::RISING) && (_output > _y1))) {
            _output = _y1;
        }
    }
    return _output;
}
```

is based on these attributes. To better reproduce the transition function, it must be evaluated at each reasonably small time point of the slope, from the initial to the final value of the expression.

The first part of the code re-computes the slope *_m* and y-intercept *_b* of the output signal if and only if the input expression changes. Then, once the initial delay has elapsed, the output signal is computed by using the slope and y-intercept previously evaluated. The transition function relies on three functions, *i.e.*, *dbl_equal*, *dbl_lequal*, and *dbl_gequal*, used to properly perform the comparison between floating point values with a customizable threshold. The complete code of such functions is shown in Listing B.1.

Let us now look at the following example, the result of the abstraction to C++:

```
if ((clock != clock_previous) && clock) {
    sample = input;
}
output = transition(sample, 0, 5E-09, 5E-09);
```

Here the *transition* function is used to drive the *output* signal with the values of the *input* signal sampled at each positive edge of a *clock*. In this case, there is no delay and both rising and falling time are set to 5*ns*. The waves generated with this code are shown in Figure 3.8.

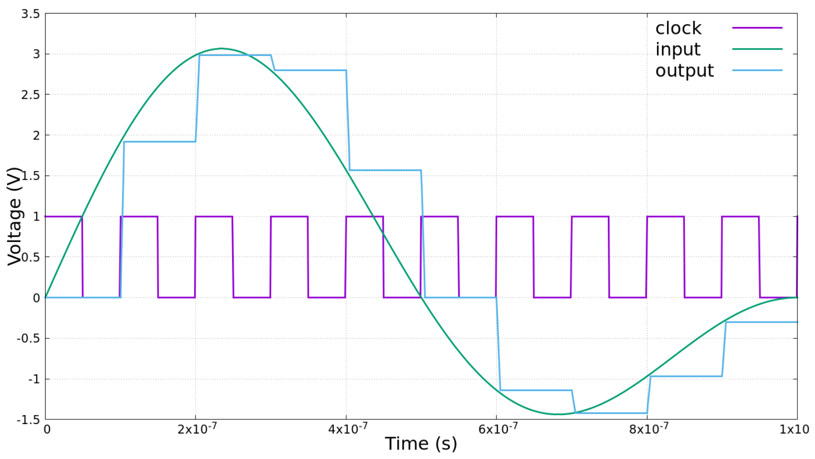


Fig. 3.8: A transition function used to track the values of an input signal at each clock event.

3.5.3.b Cross

The *cross* function is used to generate events every time an analog signal crosses the zero. The user can specify if the monitored crossing event should be a positive or negative cross. In the former case, the value goes from positive to negative, while in the latter it is the opposite. Accuracy on detecting such an event depends on the simulator which has to modify its simulation step in order to capture the cross event better. The C++ implementation of the *cross* function is shown in Listing 3.5. The arguments of the function are, the *expression* which is the current value of the signal, the *direction* of the crossing and the *functionality* to activate whenever the crossing happens.

Listing 3.5: Cross function implementation.

```
void Cross::operator()(
        analog_value_t const &          expression,
        int const &                     direction,
        std::function<void()> const &   functionality) {
    // Based on the direction, check for a crossing event.
    if (direction > 0) {
        if (positive_cross(expression))
            functionality();
    }
    else if (direction < 0) {
        if (negative_cross(expression))
            functionality();
    }
    else {
        if (positive_cross(expression) || negative_cross(expression))
            functionality();
    }
    // Extrapolate the next crossing event.
    if (detect_next_positive_cross(time_point) ||
        detect_next_negative_cross(time_point))
        _system_add_event(time_point);
    // Store the current time-point, by saving both x and y values.
    store_time_point(_system_abstime(), expression);
}
```

This function relies on two concepts, first the extrapolation of the next time-point, and second the scaling of the time-step. At each simulation step, the crossing event is checked by using the two functions *positive_cross* and *negative_cross* based on the *direction*. If the signal crosses the zero, the functionality is executed. Then, regardless of the previous check, the *cross* function controls if the next simulation step with the current time-step generates a crossing event. If that is the case, it notifies the simulator of the time point when the computed crossing event will happen. Finally, the current time point and input signal value are stored to extrapolate the next crossing event.

Let us now look at the following simple example, where the *cross* function is used to generate an asynchronous *clock* digital signal based on the analog continuous time *input* signal:

```
cross(input, 0, [&]() {
    clock != clock;
});
```

The produced waves are shown in Figure 3.9. Because the second argument is set to zero, the direction, the events are generated both during positive and negative crossings.

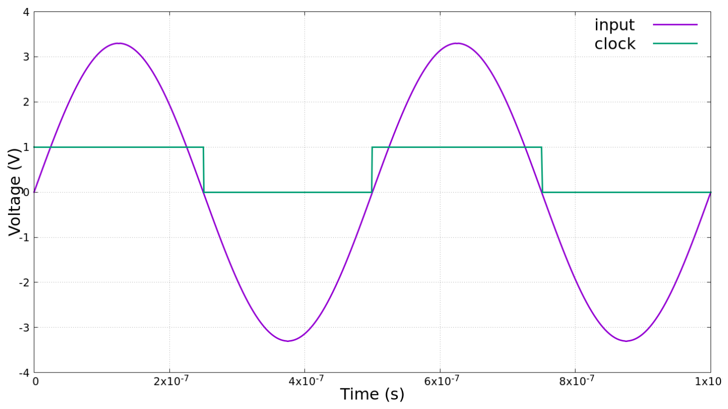


Fig. 3.9: A cross function used to generate a digital clock signal from a continuous time signal.

3.6 Transistor-level to behavioral-level abstraction

This section shows how *state-dependent small-signal* behavioral-level descriptions can replace non-linear transistor-level models. The main purpose of this abstraction is indeed speeding up the simulation of transistor-level descriptions. However, it is shown later on in Chapter 5 that an equally interesting application is the direct mapping of fault locations from transistor to the behavioral level.

Whether the objective is simulation efficiency or fault mapping, the behavioral-level description should be as accurate as possible; it must be able to emulate the correct behavior of the lower level counterpart faithfully. Once the behavioral-level description is generated, it can be abstracted with the methodology presented in the previous sections. It is clear that the equivalence heavily influences the accuracy of the final functional-level description. Furthermore, to simplify the process of fault mapping, the structure of the behavioral-level description should be as close as possible to the transistor-level one. However, this is a reasonable condition for simple circuits, for more complex ones it is hard to reproduce the behavior of transistor-level models with passive components and at the same time preserve an equivalent internal structure. In those cases, other approaches used to model the effect of faults at the behavioral-level can be adopted [43].

Let us now take the circuit of the CMOS inverter shown in Figure 3.10. The idea behind building a *state-dependent small-signal* behavioral-level description is to capture all the important states of a circuit and encode them into a set of configurations, one for each state. To build an efficient behavioral-level model which can also be abstracted to functional-level we decided to rely on linear components like resistors and capacitors. Figure 3.11 shows the topology we applied to capture the behavior of the CMOS inverter, while the equivalent Verilog-AMS behavioral level description is shown in Listing A.7. In this description, both PMOS and NMOS transistors are replaced by linear elements which can capture their non-linear behavior. The model switches between the ON and OFF states by modifying the values of these linear el-

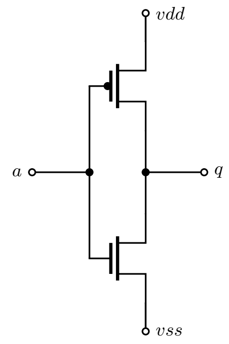


Fig. 3.10: Transistor-level circuit of a CMOS inverter.

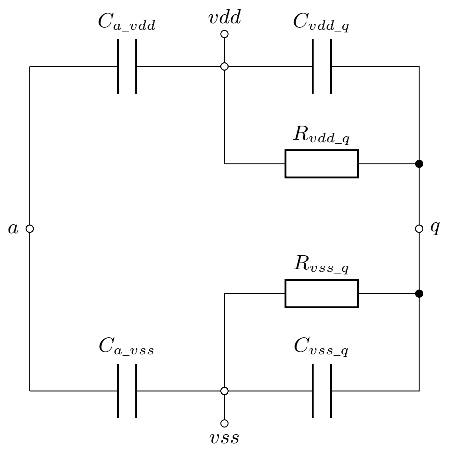


Fig. 3.11: Behavioral-level description of an inverter.

ements depending on the voltage on the input node a. Changing the state is done by using the hyperbolic tangent function. In the Verilog-AMS description, this function is called *tanhsw* and allows a smoother transition which can be adjusted by using the $smth$ variables. The input coupling of the inverter is modeled through the input capacitances, while on the output, resistors and capacitors are used to produce the inverter behavior. When a high voltage is provided to the input node a of the inverter, the value of the resistor from vdd to q is increased to model an open circuit, and the one of the lower resistor from vss to q is decreased so that the output is pulled down to vss. Conversely, when a low voltage is provided to the input node a of the inverter, the output is pulled up to vdd.

The power of this parametrized model is the ability to be tuned and characterized to fit the physical level properties of the original design (*e.g.*, based on the transistors width/length ratio). To clarify this feature, Figure 3.12 shows how the proposed abstracted description can be used to model different types of inverters, each of which implementing a different switching behavior.

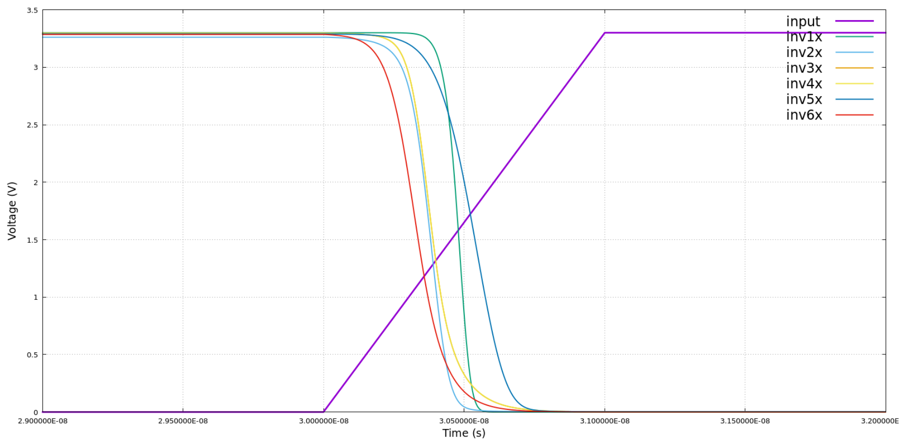


Fig. 3.12: Six different C++ CMOS inverters switching between output low and high.

Furthermore, the model proposed in Figure 3.11 is a generalization of a two-state circuit with one input and one output port, which can be manipulated to model the behavior of other non-linear transistor-level descriptions. By slightly modifying the Verilog-AMS code shown in Listing A.7 other gates can be easily generated. For instance, if we consider the structure of a *nand* gate, we have two input ports *a* and *b* instead of just *a*. As a consequence, to describe the nand gate in the same manner we just need to replicate the same two-capacitors topology for both ports. The resulting circuit for a generic two input port gate is shown in Figure 3.13.

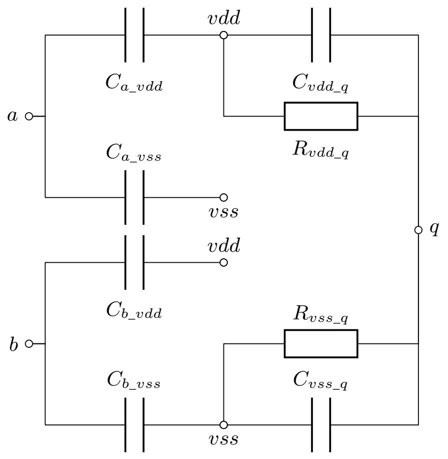


Fig. 3.13: Behavioral-level description of a generic two port circuit.

Then, we need to adjust the equation controlling the switching between its states (which is at line 42 in Listing A.7):

```
x = min(V(a), V(b));
```

To model a *xor* gate the line must be changed into:

```
x = max(V(a), V(b));
```

An *exnor* gate can be modeled in the same way by using the following behavior:

```
x1 = max(V(a), V(b)); // a or  b
x2 = min(V(a), V(b)); // a and b
x1_u = tanhsw(x1, vth_u, smth_u);
x1_d = tanhsw(x1, vth_d, smth_d);
x2_u = tanhsw(x2, vth2_u, smth2_u);
x2_d = tanhsw(x2, vth2_d, smth2_d);
// not(a or b) or (a and b)
x_u = max(0, min(1, (1 - x1_u) + x2_u));
x_d = max(0, min(1, (1 - x1_d) + x2_d));
```

To properly capture the *exnor* behavior we needed to duplicate the pair of variables controlling the voltage threshold and smoothing factor (*i.e.*, vth and $smth$), as well as adding new support variables. Increasingly complex behaviors require of course more complex switching functions, more linear elements and most probably need to take into account more states. To exemplify this aspect, let us take the transistor-level description of the CMOS analog switch shown in Figure 3.14. In this model,

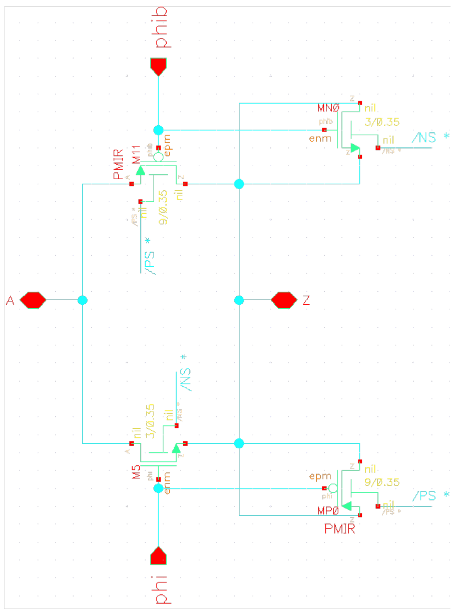


Fig. 3.14: Transistor-level description of a CMOS switch.

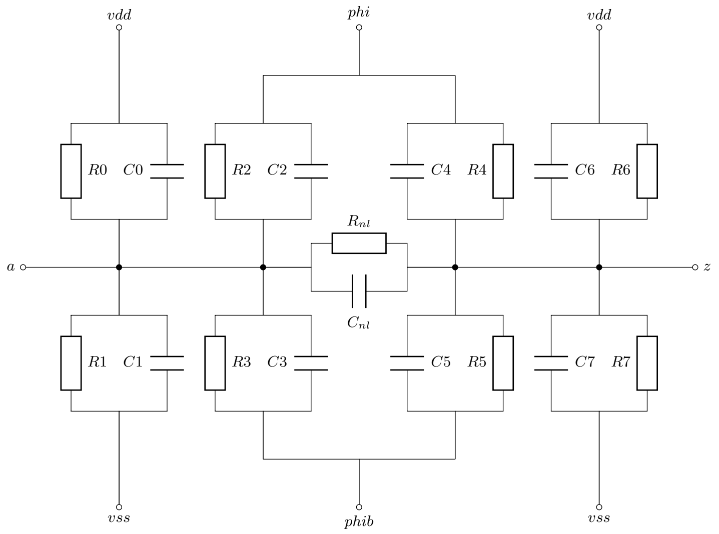


Fig. 3.15: Behavioral-level description of a switch.

the switching is controlled by the two analog ports `phi` and `phib`. The signals called `PS` and `NS` are actually `vdd` and `vss` respectively. The equivalent behavioral-level circuit of the switch is shown in Figure 3.15. To cover all the physical aspects of the original switch circuit we require a total of 18 linear components. Furthermore, we need to store a total of four configurations for the linear elements, since there is a configuration for each one of the *four* combinations of the two control ports.

It is worth noting that in most of the analog descriptions, components parameters could be replaced in the corresponding equations inside the design. However, with *state-dependent small-signal* behavioral descriptions, components parameters cannot be replaced by their actual values, because they need to change during the simulation to emulate all the states of the model. If we consider the behavioral switch model, the symbolic analysis has to deal with a total of 18 symbols, becoming a complex task which could take a considerable amount of time. For this reason, the analog abstraction methodology must be extended to enable a better scaling with the number of symbols inside the design.

3.7 Behavioral-level interface building

In the previous chapter, and in Section 3.4 we introduced the analog abstraction methodology, which generates symbolic transfer functions of the circuit behavior at each of its output ports. We showed that for each conservative sub-circuit, it sets up the system of equation for symbolic analysis. Whenever possible it replaces constants, and eventually even parameters, to reduce the complexity required to solve the

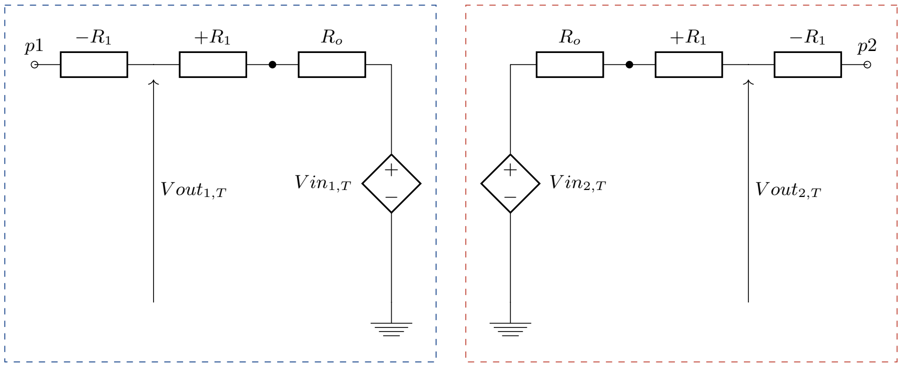


Fig. 3.16: Two-port waveform-relaxation interface.

system. Then, the resulting set of equations is used to generate the functional-level description written in C++.

The symbolic analysis of large analog integrated circuits has an essential role in the study of the circuit behavior. It can be a useful tool for testing analog and mixed-signal systems during, and after, the design process of VLSI ASICs [44]. Unfortunately, this kind of analysis is a complex task because of the difficulties in handling large symbolic formulas [45]. For example, the inverter we have shown in Figure 3.11 is modeled with six symbols (*i.e.*, capacitance and resistance values), while the switch shown in Figure 3.15 requires 18 symbols. All these symbols inevitably increase the complexity of solving the system of equations. Furthermore, if the inverter circuit is connected through a conservative node to other circuits, the complexity increases even further.

To effectively overcome this scalability problem and also to allow composing already abstracted models, the proposed methodology relies on a special type of interfaces. Such interfaces rely on the Waveform Relaxation (WR) theory, a numerical algorithm allowing to split a system of Ordinary Differential Equations (ODEs) into subsystems which can be solved separately [46, 47, 48]. In this work, the circuit implementing the WR principles is used as an interface between the macro-blocks. Its structure is shown in Figure 3.16, where two interfaces are used to drive two separated sub-circuits.

Focusing on the left-hand side circuit, connected in series with the port $p1$ of the first sub-circuit there are three resistors and a Voltage Controlled Voltage Source (VCVS). The first pair of resistors ($+R1$, $-R1$)) allows to convert the continuous time formulation in terms of *voltage/current*, to a pair of voltage waves ($Vin_{1,T}$, $Vout_{1,T}$) in discrete-time with sampling period *T*; where the first is a wave coming from the outside, and the second one is the output wave. The third resistor R_o represent the outside circuit inside the interface of the block. The right-hand side circuit is a mirror of the left one. During simulating, at the end of each time-step, the voltage waves are used to drive the controlled sources. The value of $Vout_{1,T}$ is assigned to $Vin_{2,T}$, and vice-versa $Vout_{2,T}$ is assigned to $Vin_{1,T}$. This formulation inspired

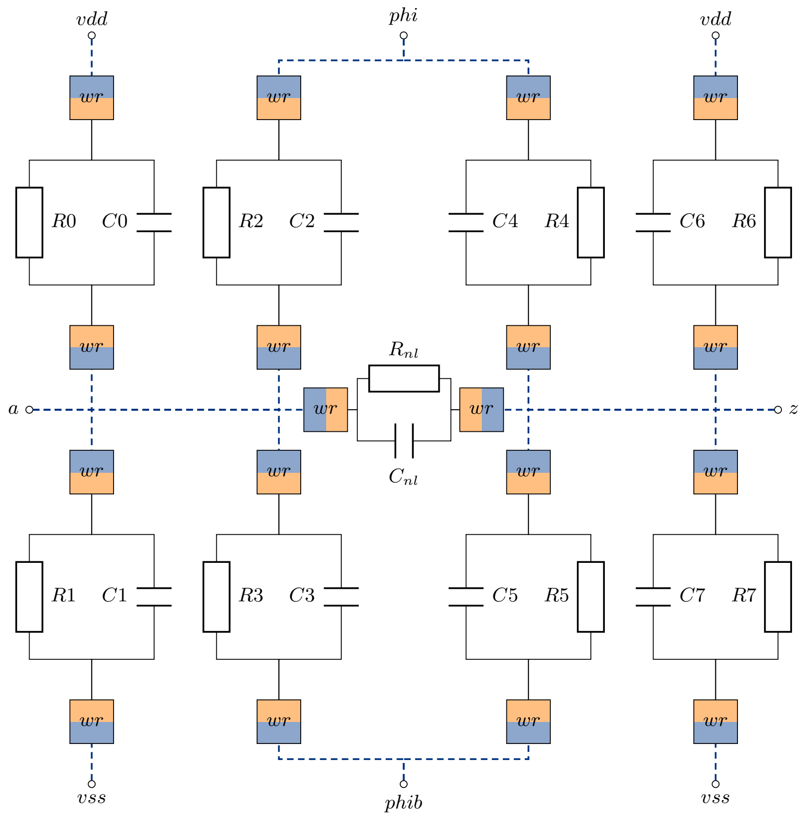


Fig. 3.17: Switch circuit with multi-port waveform relaxation interfaces. The blue side of each interface represent the discrete-time waves. The orange side represent the continuous-time analog circuit.

from transmission-line theory, the interface correspond to a loss-less transmission line with a characteristic impedance R_o and delay T. Provided the delays through the interface are small enough the system will be stable if the system it models is stable [49].

The circuit shown in Figure 3.16 can be used to connect just two nodes, an extension is required to be able to connect more nodes. For instance, the work in [50] shows a VHDL-based interface called “MiXED”, which defines a type of controlled source which computes not only the output voltage but also the impedance, allowing the composition of multiple “MiXED” sources. Inside the delivery D3.2.2 of the SMAC project [49], a multi-terminal waveform relaxation interface is proposed, allowing to connect multiple interfaces together.

Now, if we consider the behavioral description of a switch shown in Figure 3.15, we would place the interfaces on the output pins, *i.e.*, pins *a*, *z*, *phi*, *phib*, *vdd*, and *vss*. This solution solves the problem of composing more than one switch; however, we need to consider that each interface introduces a new symbol, *i.e.*, the variable controlling the VCVS. In that case, the equations system of the switch circuit with the WR interfaces would have a total of 24 symbols. Solving the system produces sensibly large symbolic equations, which can have a repercussion on the efficiency of the generated model. When dealing with relatively large macro-blocks (*i.e.*, in terms of symbols), a possible solution is splitting the circuit of the block by using the WR interfaces internally. The resulting circuit is shown in Figure 3.17.

The circuit shown in the figure also presents a peculiar configuration, where multiple WR interfaces are connected. The orange side represents the continuous-time analog circuit, the one depicted in Figure 3.16. The blue side represents the discrete-time waves exchanged between interfaces, while dashed lines connect the interfaces which are exchanging values. During the simulation, all the connected waves are used to drive the same group of interfaces and their VCVSs. Give a group of N_g interfaces, *e.g.* the ones connected to port *a* in Figure 3.17, each VCVS is driven by the result of this equation:

$$Vout_T = \frac{\sum\limits_{n=1}^{N_g} Vout_{n,T}}{N_g} \tag{3.1}$$

which is the mean beween the values of the driving waves.

3.8 Holistic platforms for Industry 4.0

The concept of Industry 4.0 [51] represents an innovative vision of what will be the factory of the future. The principles of this new paradigm are based on interoperability and data exchange between different industrial equipments. Cyber-Physical Systems (CPSs) cover one of the main roles in this revolution. Each machine inside the production line can be modeled as a CPS, while the entire factory can be seen as a system of systems also known as Cyber-Physical Production System (CPPS). This CPPS represents the so called *Digital Twin* of the factory, and allows performing analysis regarding the real factory [52]. The interoperability between the real industrial equipment and the *Digital Twin* [53] allows making predictions concerning the quality of the products. Moreover, an accurate *Digital Twin* would make possible to estimate *mechanical wear* and *aging* of industrial equipments.

Several tools [54] allow modeling a production line, considering different aspects of the factory, *i.e.*, geometrical properties and information flows. However, these simulators do not provide natively any solution for the design integration of CPSs. The resulting production line models are not accurate enough, as a result having precise analysis concerning the real factory is almost impossible. The idea, summarized in Figure 3.18, is to harness the power of the holistic platform generated with the work of this thesis for integrating an entire CPS inside a production line simulator. The approach relies on the abstraction and mixed-signal scheduling techniques presented in this chapter and a production line simulator called *Siemens Plant Simulation*.

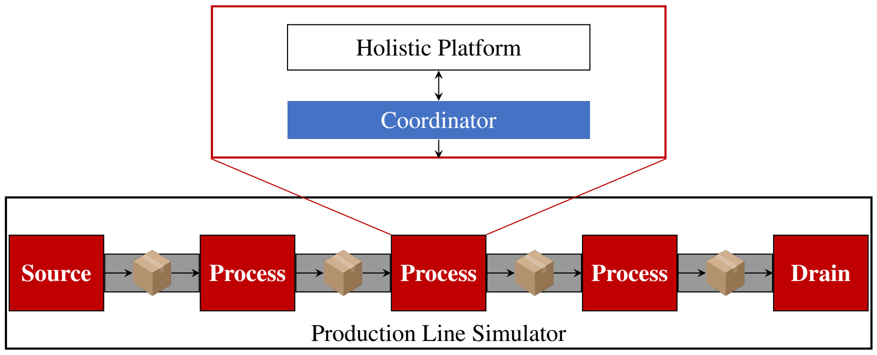


Fig. 3.18: Overview of the approach for the integration of a CPS integration in a production line simulator.

A *production line* is the composition of processes, organized into a chain, which has the main purpose of handling information, *e.g.*, geometric properties, processing time, energy consumption, and failure rate. Simulating such production systems allows making strategic decisions, which aims at optimizing the entire production line under different aspects, *e.g.*, cost, quality and productivity. In the past few years several providers proposed different tools for modeling manufacturing processes [54, 55]. Report like the one proposed in [55] summarizes the main characteristics of all the newest tools periodically.
Regardless of the difference between each tool, all of them share the following principles:

- *Layout Planning*: Represents the geometrical structure of the production line. All simulators have a library of components (*i.e.*, generic processes, assembly stations, and buffers) which allow to model the factory, by considering physical constraints.
- *Material Flow/Fluid Simulation*: Represents the carrier of information, the movements of products from a process to the others. This is made possible with components like line transporters or pipe, depending on the material state of matter, *i.e.*, solid or fluid.
- *Process Simulation*: Represents the physical transformation made by the equipment of the factory to the products.

From the simulation perspective, most of the available simulators rely on the *discrete-event* model of computation. Correlating this Model of Computation (MoC) and production lines is reasonably straightforward. For instance, when a product enters or exists a process, an event is triggered and the specific equipment can execute its relative action. The study presented in this section makes use of Siemens *Plant Simulation*, a simulator which over the years has become a standard de facto for designing production lines, with an intuitive and easy to use environment. It is a model-based tool that provides a library of customizable components that represent the basic building blocks of a factory. Combining these blocks and most importantly

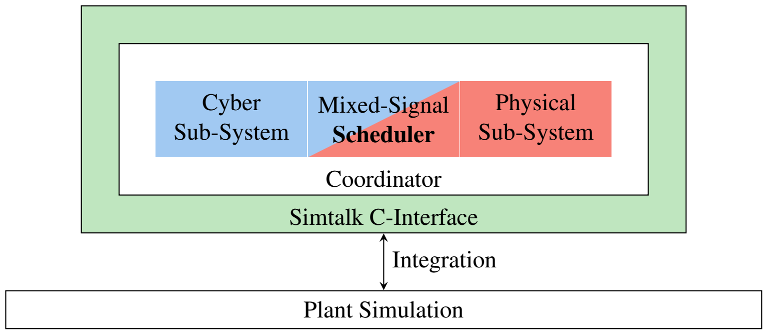


Fig. 3.19: Abstract-based Experiment Setup

specifying the properties associated with each one of them, allow modeling different aspects of a production line. The products are called Mobile Unit (MU) and they represent the entities moving among the blocks of the production line.
The fundamental blocks provided by the tool are:

- `Source`: generates MUs and it is the entry point of a production line;
- `Drain`: the final block of the line, it is the exit point for the MUs;
- `SingleProcess`: it represents the physical process of an equipment on an MU;
- `Line`: it represents the equipments that transport an MU from a block to another;

Plant Simulation offers the possibility to customize the behavior of every block with an internal programming language called *SimTalk*. This proprietary language allows defining methods which can manipulate the elements of the production line. But most importantly, these methods can be linked as *call-backs* to events happening on the line, so that they are triggered whenever such events occur. Every building block provided by the library can be controlled through a *SimTalk* method with a set of event controllers (*i.e.*, a MU entering or exiting a block). It also provides a functionality called *C-interface* to import dynamic libraries written in C/C++, for customizing the behavior of the simulation or connect external tools.

Figure 3.19 depicts the structure used to integrate the holistic platform inside *Plant Simulation*. The approach applies the two abstraction methodologies, for digital and analog models, and the mixed-signal scheduling. Once applied to the CPS description, the Cyber and Physical abstracted models need to be integrated into *Plant Simulation*. This task is performed by a coordinator which synchronizes the CPS and *Plant Simulation* at each analog simulation step. The resulting CPS is then integrated in *Plant Simulation* using *SimTalk* C-Interface APIs.

3.9 Experimental results

This section shows the effectiveness of the proposed methodology on a number of case studies of increasing complexity. The experiments are performed on a 64-bit linux machine, with 16 GB of memory and an Intel(R) Core(TM) i7-3770 CPU

@ 3.40GHz. A commercial SPICE-based simulator is used to simulate both the transistor-level circuit written in SPICE and the behavioral-level description written in Verilog-AMS. The extension to the abstraction methodology is implemented inside the same tool presented in the previous chapter, the Analog Systems Translation and absRAction tooL (ASTRAL) tool.

The main purpose of the methodology shown in this chapter is the integration of analog *multi-domain* components inside complex virtual platforms, to simulate realistic scenarios. For this reason we selected a complex virtual platform that can be used to interact with the analog components, as in the running example shown throughout the chapter. The virtual platform used for the experiments is the SMAC SMAC Smart System Test Case (S3TC) [40]. Its structure is depicted in Figure 3.20. This complex design includes a MIPS CPU and a memory bank to perform control and computation, connected to a set of digital peripherals through an Advanced Peripheral Bus (APB), designed specifically for low bandwidth control accesses based on the Advanced Microcontroller Bus Architecture (AMBA) standard. The program running on the CPU can communicate with the analog components through a series of DACs and ADCs.
The remainder of the section is structured as follows:

- Section 3.9.1 shows how the methodology can be applied to different physical disciplines and modeling styles;
- Section 3.9.2 shows the application of the methodology to a complex case study.
- Section 3.9.3 shows how the mixed-signal scheduling applied after the abstraction can sensibly reduce the synchronization overhead.
- Section 3.9.4 shows the results of the abstraction from transistor to behavioral level.
- Finally, Section 3.9.5 presents the results for the integration of the abstracted platform inside the plant simulator.

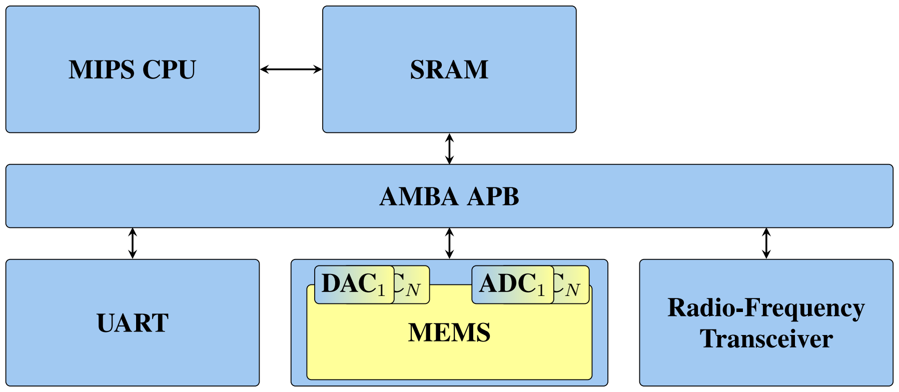


Fig. 3.20: Heterogeneous virtual platform where the MEMS component is inserted into.

Table 3.3: Characteristics of the selected benchmarks. The used disciplines are: (1) Logic, (2) Electrical, (3) Rotational, (4) Magnetic, (5) Kinematic, (6) User defined discipline.

	Disciplines							Lines of	Number of	Number of	Abstraction
Benchmark	(1)	(2)	(3)	(4)	(5)	(6)	Domain	Code	Variables	Equations	Time (*s*)
Low Pass Filter		✓					Laplace	26	2	1	0.009
Magnetic Winding		✓		✓			Time	25	10	2	0.010
Motor		✓	✓				Time	41	12	2	0.009
Bouncing Ball	✓					✓	Time	64	5	5	0.009
Mass-Spring-Damper					✓		Time	98	8	4	0.007
Vibration Motor	✓	✓	✓				Time & Laplace	533	38	42	0.064
Pico-Projector	✓	✓	✓			✓	Time & Laplace	546	41	46	0.084

3.9.1 Multi-domain abstraction evaluation

Table 3.3 summarizes the main characteristics of the multi-domain benchmarks used as case study. Considered models are expressed by using the (1) Logic, (2) Electrical, (3) Rotational, (4) Magnetic, (5) Kinematic, and (6) User defined disciplines. For each model, it is reported the specification domain (*i.e.*, Laplace- or Time-domain), the number of lines of code (LoC) of the original description, the number of continuous variables, the number of equations, and the time required to abstract the model to C++. The first five benchmarks come from a repository of open-source models[1]. They are used to show the applicability of the methodology to different physical disciplines and modeling styles. The *vibration motor* refers to the running example used throughout this chapter. While the *pico-projector* is a realistic case study presented in the next section, used to show the methodology effectiveness.

Table 3.4 reports the time need to simulate the benchmarks for *1 second* of simulated time and a simulation step of *50 nanoseconds*. The table compares the efficiency of the original Verilog-AMS and the abstracted C++ code in two configurations: the benchmark alone and integrated inside the virtual platform S3TC. The comparison between the simulations of the component without the platform highlights specifically the impact of the abstraction.

3.9.2 Industrial case study: pico-projector

The previous experiments showed the applicability of the methodology to some synthetic examples. This section shows the application of the methodology on a more complex case study. It is an industrial device, comprehending a multi-discipline MEMS, integrated within a complex hardware platform able to perform computation and communication. The MEMS is the micro-mirror of a pico-projector[2]. The pico-projector is based on three laser sources for the primary colors (*i.e.*, red, green and blue). These are modulated and their light converges into a single beam deflected

[1] www.designer-guide.org

[2] The design has been kindly furnished by STMicroelectronics, in the context of SMAC (SMArt systems Codesign) European Project (grant number: FP7-ICT-2011-7-288827)

Table 3.4: Simulation times needed to simulate the selected benchmarks and achieved speed-up.

Benchmark	Verilog-AMS		C++			
	Component	Platform	Component		Platform	
	time (s)	time (s)	time (s)	speed-up (x)	time (s)	speed-up (x)
Low Pass Filter	3493.49	5268.20	2.07	1687.67	153.46	34.32
Magnetic Winding	3477.08	5197.33	1.89	1839.72	154.43	33.65
Motor	2281.46	3762.47	2.66	857.69	151.68	24.80
Bouncing Ball	2173.82	3535.51	0.85	2557.43	149.77	23.60
Mass-Spring-Damper	3653.10	5345.86	1.57	2326.81	148.71	35.94
Vibration motor	6235.12	7810.64	59.96	103.98	207.13	37.70

by a micro-mirror. This mirror is moved by drivers, in accordance to the decisions taken by a digital board.

Figure 3.21 depicts the components of the MEMS system moving the mirror of the pico-projector, their characteristics and how they are connected to each other. Follows a detailed description of each component:

- The *digital board* implements the control logic for the mirror, and generates the clock signal synchronizing the DACs and ADC. It generates commands for the actuators, and gather data sensed about the motion.
- Two DACs components convert the digital signals generated by the digital board into voltage values. The model is linear and based on the *logic* and the *electrical* disciplines.

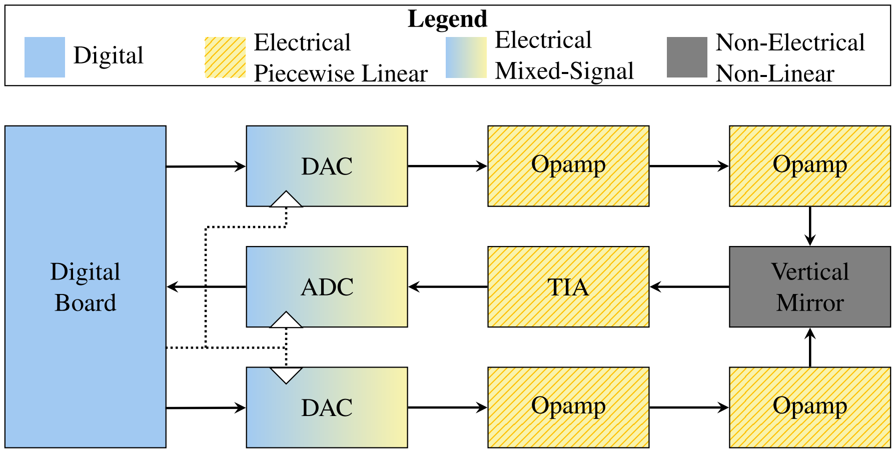


Fig. 3.21: Structure and classification of the components in the Vertical mirror system.

- Each DAC generates a voltage value that is used as input for two OpAmps in series, similar to those used in the vibration motor running example.
- The *vertical mirror* component models the dynamic of one of the two projector mirrors. It uses the *electrical* discipline to be interfaced with the two OpAmps. In this model two rotational discipline are used: the *rotational_velocity* and the *rotational_acceleration* disciplines. The first discipline is standard (as shown in Table 3.2), the latter is a user-defined discipline relating angular force and acceleration. The equation system is non-linear: it employs polynomial functions, implemented using the *logic discipline*.
- A current flow is generated by the mirror to feedback motion information to the digital board. The TransImpedance Amplifier (TIA) receives the generated current flow and convert it into a voltage value. It is modeled using the *electrical discipline* and sets a threshold to the amplification, thus it is piecewise-linear. The discontinuities are specified using the *logic discipline*.
- The ADC converts the voltage generated by the TIA into a digital value for the board. It is represented by a linear model based on the *electrical and logic disciplines*.

The components are designed independently and aggregated to obtain the final system. The feedback loop between mirror and digital board is necessary to evaluate and compensate the *ripple effect* caused by the steering mechanisms and affecting the pico-projector. The ripple can be mitigated or even eliminated by using a compensation algorithm implemented as a SW running on the digital board. The components are modeled in Verilog-AMS, except for the board that has been implemented in Verilog. The virtual prototype can be used to develop control strategy and functional validation. To do so, the component can be simulated by using any SPICE-based simulator, capable to perform also HDLs simulation (*e.g.*, Mentor's Questa ADMS). However, such a simulation environment needs to employ different technologies and therefore it introduces heavy synchronization overhead [5]. Thus, abstraction techniques come in handy to reach faster simulation and therefore faster validation.

Figure 3.22 depicts the result of the *discipline analysis* applied to the pico-projector. Six different disciplines have been used. In particular, the design is partially modeled by using a custom rotational discipline: it describes the angular acceleration, rather than velocity, of the rotating mirror. Six different electrical sub-models (red dashed lines) have been isolated. The four OpAmps in the models are four instances of the same model, instantiated by using different parameters to customize gains and thresholds.

As shown in Section 3.4.1, part of the OpAmp belongs to the *frequency domain* because of the *Laplace* function, and it also contains *piecewise linear behaviors* due to the thresholds on output values modeled by `if-then-else` statements. All of these aspects are dealt with by using the methodology described in the previous sections. Similarly, the TIA component has a structure similar to the one of the OpAmp, as such, it contains both *frequency domain* functions and has a *piecewise linear behavior*. Its code is shown in Listing A.5. An interesting aspect of the *vertical mirror* model si the presence of the polynomial analog function written in Verilog-AMS shown at the top of the Listing 3.6. The application of the strategy shown in Section 3.4.4 to the polynomial function yields the C++ equivalent function shown at the bottom of Listing 3.6.

Listing 3.6: Polynomial function inside the *vertical mirror* deisgn, specified in Verilog-AMS (*above*), and the corresponding C++ implementation (*below*).

```
analog function real polyfit_4;
    input c0, c1, c2, c3, c4, t;
    real  c0, c1, c2, c3, c4, t;
    polyfit_4 = c0 + c1 * pow(t, 1) + c2 * pow(t, 2) +
                     c3 * pow(t, 3) + c4 * pow(t, 4);
endfunction;
```

```
#include <cmath>
inline double polyfit_4(double c0, double c1, double c2, double c3,
                        double c4, double t) {
    return (c0 + c1 * pow(t, 1) + c2 * pow(t, 2) +
                 c3 * pow(t, 3) + c4 * pow(t, 4));
}
```

Table 3.5 reports the time need to simulate the pico-projector for *1 second* of simulated time and a simulation step of *50 nanoseconds*. The C++ component by itself can reach a 80*x* of speed-up, while this improvement decreases when integrate inside the S3TC. Furthermore, the speed-up achieved with the pico-projector is lower than the one achieved with the experiments shown in the previous section. This is primarily due to the partitioning of the circuits and their systems of equations. The previous examples have a coarser partitioning that allows better simplifications

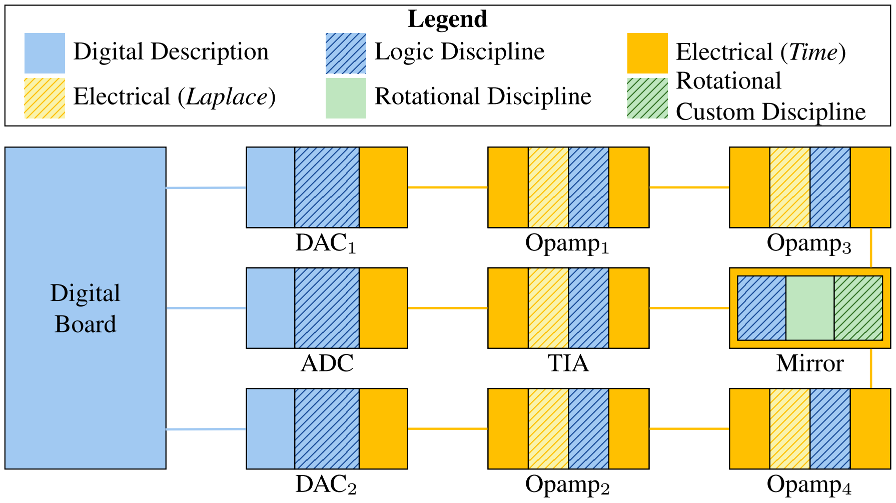


Fig. 3.22: Structure and classification of the components in the Vertical mirror system.

Table 3.5: Simulation times needed to simulate the pico-projector benchmark and achieved speed-up.

Benchmark	Verilog-AMS		C++			
	Component	Platform	Component		Platform	
	time (s)	time (s)	time (s)	speed-up (x)	time (s)	speed-up (x)
Pico-Projector	9067.63	10156.27	112.43	80.65	276.62	36.71

Table 3.6: Benchmarks characteristics and time required for the automatic abstraction.

Benchmark	Lines of Code	Number of Variables	Number of Equations	Abstraction Time (s)
Voltage Limiting Operational Amplifier	33	2	2	0.007
Ideal Operational Amplifier	31	12	6	0.009
Transimpedance Amplifier	33	5	3	0.007
MEMS Mechanical Actuator	546	41	46	0.084

during the sub-modules abstraction. The pico-projector instead, has a fine grained partitioning as depicted in Figure 3.22, leading to small portions more difficult to be optimized. These small parts must be reconnected to each other with communication mechanisms which introduce overhead. The pico-projector also requires to solve polynomial and complex non-linear functions (*e.g.*, Listing 3.6) at run-time that cannot be removed if the original behavior of the abstracted component must be preserved.

3.9.3 Mixed-signal scheduling for system integration

Table 3.6 reports for every used benchmark: number of lines of code, number of internal variables, number of equations belonging to the analog system, and the time required to generate the abstracted version. All of these models have been chosen from an online repository of Verilog-AMS models[3], the designs released by Accellera [10], and from industrial partners. In particular, the MEMS mechanical actuator is a MEMS design provided by two industrial partners in the context of the SMAC European Project (FP7-ICT-2011-7-288827).

Table 3.7 reports the simulation time needed to simulate both the analog components by themselves (*i.e.*, *Component* columns) and once they have been integrated into the smart system (*i.e.*, *Platform* columns). They have been simulated by using their original description (*i.e.*, *Verilog-AMS* in the heterogeneous implementation). Then, they have been automatically translated into *SystemC-AMS* by using the approach presented in Chapter 2, and simulated both alone and integrated within a

[3] www.designers-guide.org

Table 3.7: Execution times for different abstractions simulated both alone and together with the smart system.

	Heterogeneous (Verilog-AMS)		*ABACUS* automatic translation (SystemC-AMS/ELN)				*ASTRAL* automatic abstraction (C++)			
	Component	Platform	Component		Platform		Component		Platform	
Benchmark	time (s)	time (s)	time (s)	speed-up (x)	time (s)	speed-up (x)	time (s)	speed-up (x)	time (s)	speed-up (x)
Voltage Limiting Operational Amplifier	3138.62	4712.55	Not applicable due to nonlinearities				3.62	867.29	159.73	29.50
Ideal Operational Amplifier	3499.48	5024.72	30.21	115.83	1599.52	3.14	1.77	1982.30	156.42	32.12
Transimpedance Amplifier	3438.67	5114.33	Not applicable due to nonlinearities				1.91	1801.78	154.92	33.01
MEMS Mechanical Actuator	8078.35	9769.48	Not applicable due to nonlinearities				114.64	70.47	275.53	35.46

SystemC model of the S3TC platform. Finally, the models have been abstracted and then integrated into the C++ implementation of the S3TC by using the proposed scheduling approach.

The results reported in the table highlight the approach effectiveness both concerning the stand-alone simulation of the analog components and the simulation of the entire virtual platforms. For each component, it provides up-to three order of magnitude speed-up *w.r.t.* the SPICE-based simulation. Furthermore, it always outperforms the translation approach.

The comparison between simulations of the entire platform aims at showing the effectiveness of the scheduling approach. While the translation approach provides only marginal speed-up to the platform simulation, the abstraction methodology developed in this thesis allows to speed up the simulation of more than one order of magnitude. The variability of resulting speed-up is due to the different synchronization imposed by the application scenarios used to test the different designs. These different scenarios lead to different synchronization patterns between the digital and the analog part of the platform. Thus, different impact of the synchronization overhead among different designs.

Figure 3.23 shows the different overhead contributions due to three major categories: simulation of the *component*, the digital part of the *platform* and the *synchronization* between the two. For each component, the overhead contributions are depicted for both the heterogeneous and the abstracted homogeneous platform implementation, called *Mixed* and *C++* respectively. The y-axis depicts in logarithmic scale the contribution in terms of simulation time for each category. In a SPICE-based execution, the synchronization represents a bottleneck which has a sensible impact on the simulation time. In the abstracted models, the time taken to synchronize the digital platform and the analog component is less impacting on the overall execution, thus removing the bottleneck. The effectiveness of the proposed approach can be more easily appreciated when complex analog components are connected to the platform. As for the case of the *MEMS Mechanical Component*, where the synchronization is lower than both component and platform.

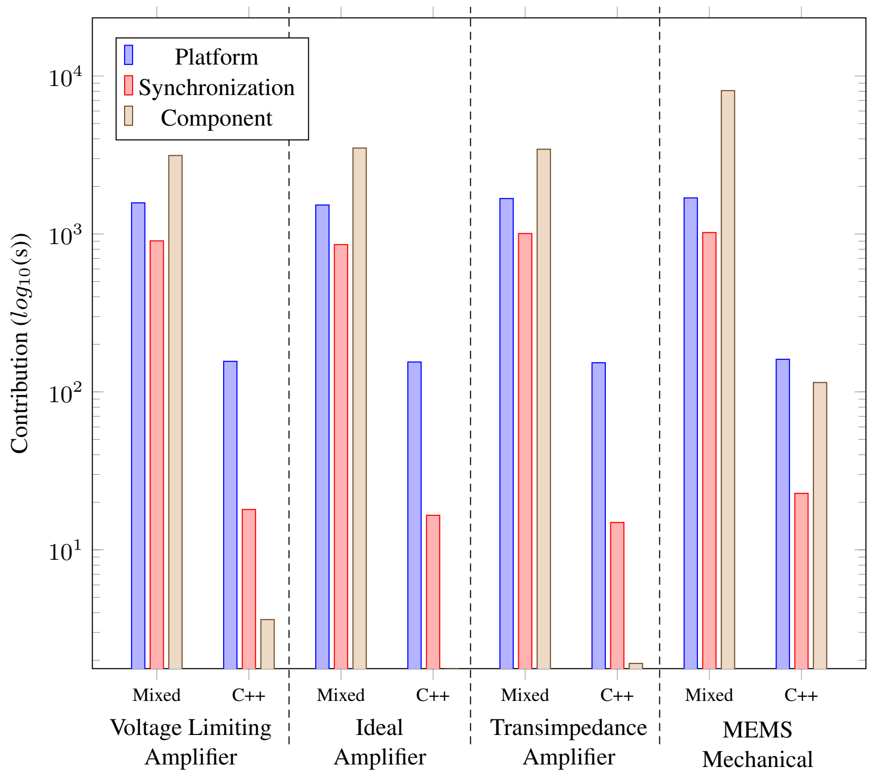


Fig. 3.23: Simulation overhead contributions (expressed in seconds on a logarithmic scale) for the different parts of the system (*i.e.*, analog component, digital platform and synchronization) for the *heterogeneous virtual platform* (Mixed) and the *homogeneous virtual platform* (C++).

Figure 3.23 also highlights the impact of the methodology on the synchronization overhead. The abstraction approach generates discrete event models for both the analog and digital portions of the platform. Thus, it is possible to employ a lighter synchronization mechanism based on function calls rather than more burdensome inter-process communication. Furthermore, the temporal decoupling technique allows to generate lighter communication schema between the analog and digital subparts of the system. This further reduces the overhead introduced by synchronization.

Finally, the abstracted modules accuracy has been evaluated by considering the normalized root-mean-square error (NRMSE) of their output compared to the Verilog-AMS output. For all the considered components, the NRMSE is ranging from 10^{-5} to 10^{-8} when the components are simulated alone, thus highlighting a high level of accuracy.

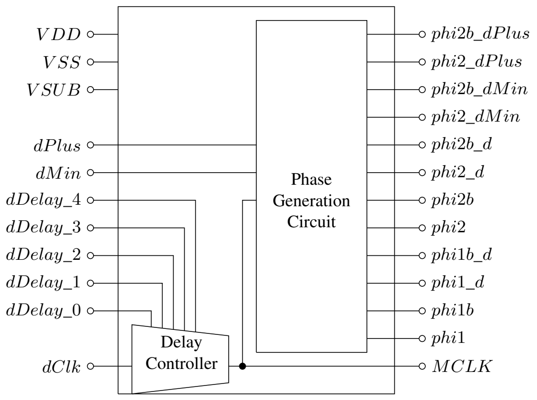


Fig. 3.24: Interface of the clock generator.

3.9.4 Transistor-level to behavioral-level abstraction and interface building

In this section, we show the results achieved by applying the abstraction from transistor to behavioral-level to the case study of a *clock generator* written in Spectre.

Its structure is shown in Figure 3.24 and contains *352* components for a total of *626* transistors. This model generates a set of signals pairs namely *phi* and *phib*, which are used to control the input sampling operations of a sigma-delta ADC. Each pair of signals is used to activate several switches like the one shown in Figure 3.15 inside the ADC. The *dClk* port is the reference clock, with a width of 30*ns*, a period of 60*ns*, and a rise/fall time of 1*ns*. The set of *dDelay* ports controls the delay applied to the reference clock to produce *MCLK*. All the other output signals are generated by the *Phase Generation Circuit* taking as input *MCLK*, *dPlus*, and *dMin*.

Follows the process with which the *clock generator* has been abstracted to the functional level. First, each building block of the Spectre description has been replaced with the equivalent behavioral-level description written in Verilog-AMS. Then, a copy of the WR circuit shown in Figure 3.16 is connected to every port of the building blocks. The abstraction methodology presented in Section 3.4 is applied to each macro-block to produce functional-level C++ descriptions. Finally, the clock generator is re-composed by connecting together the WR interfaces. Table 3.8 compares the simulation time required to simulate 1*us* at transistor-level, at behavioral-level with the *state-dependent small-signal* Verilog-AMS models, and finally the functional-level C++ description. The speed-up gained with the C++ description is extremely promising, especially because the current C++ version relies on a simulator or simulation model which has no history before the work of this thesis. Furthermore, there are more improvements that can, and will, be implemented in the future, from a smart dynamic simulation step to code optimizations.

Table 3.8: Simulation time required to perform a complete fault simulation of the clock generator for a total of thirty faults.

Abstraction Level	Simulation Time (s)	Speed-Up (x)
Transistor	106.70	*reference*
Behavioral	24.48	4.35
Functional	13.71	7.78

3.9.5 Holistic platforms for Industry 4.0

The integration of the holistic platform inside *Plant Simulation* approaches has been tested with a real use case scenario of a simple production line. The production line is composed of three-processes and represents a bending operation of metal sheets, exemplified in Figure 3.25. The three steps are represented using *Plant Simulation* `SingleProcess` blocks. The first process applies to the metal sheet a bar-code containing the information of the desired bend angle. The second represents the bending machine that reads the angle to bend from the bar-code previously applied to the metal sheet and executes the bending operation. The real bending process of the equipment requires around 1.25 seconds to bend a metal sheet. The last process represents the quality check stage. The process compares the desired angle read from the bar-code and checking real bent metal sheet. This process redirects the metal sheets in three different boxes depending on their quality.

The approach presented in this work integrates a CPS inside the bending process of the production line, to make a more accurate estimation of the production in terms of productivity and quality. The *Physical* system represents the behaviour of a bending machine, described using Verilog-AMS, that is controlled by the *Cyber* system. The *Cyber* system of the CPS is a hardware platform composed of a CPU, a memory, a bus, a bar-code reader and a mechanical actuator used to bend metal sheets. All the components of the hardware platform are described by using Verilog or VHDL languages at RTL. The bending control software is cross-compiled and then stored in the model of the memory. The firmware reads the angle to bend from the metal sheet using the bar-code Sensor, and then redirects it to the *Physical* systems with a set of commands. When the the bending operation is completed, *Cyber* system is notified.

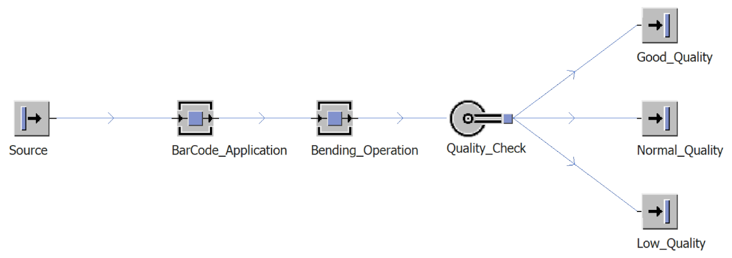


Fig. 3.25: Metal Bending production Line modeled with *Plant Simulation*

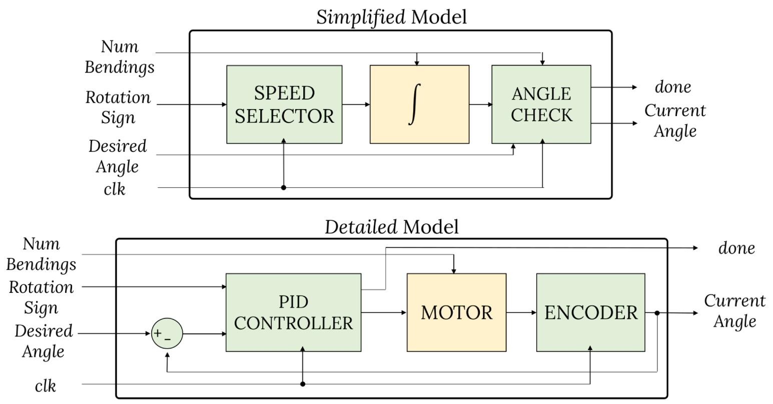


Fig. 3.26: Simplifed and detailed physical models.

Finally, the control software releases the bent metal sheet and sets the machine for a new bending operation.

Two version have been developed for the *Physical* part of the CPS: *Simplified* and *Detailed*. Their structure is shown in Figure 3.26.

In the *Simplified* model, the *Physical* system consists of a behavioral description of the bending equipment, with a low level of detail. The *Physical* system is modeled principally with an integrator that describes the bender operation, which receives as input a value that represents the constant bending speed to apply. This value is calculated by a *Speed Selector* node, which adopts two different values according to the bending rotation versus. The value provided by the integrator is then compared with the desired angle by the *Angle Controller* node in order to drive the *done* signal. Both the *integrator* and the *Angle Controller* receive as input the *number of bendings* which alter, respectively, the bending speed and the final angle. This allows simulating the machinery wearing affects.

The *Detailed* model is a functional-level description of the *Physical* system that includes a DC motor model. This level of detail allows making a more precise estimation of the execution time and the quality of operation. A *PID Controller* is used to control the DC Motor position in order to bring every sheet to the desired angle, which is read by an encoder that translates the motor rotational position in a digital value. The *PID Controller* takes also care of driving the *done* signal in order to notify the end of every bending process. The DC Motor model uses the information about the *number of bendings* to modify its internal parameters: this allows to simulate its wearing since those values represent the mechanical and electrical characteristics of the Motor.

The *Coordinator* is connected to the CPS simulator, while on the other side, the *Coordinator* is integrated in *Plant Simulation* using *C-Interface* proprietary interface. This allows to integrate it as an external dynamic library (`.dll`) directly into the production line simulator. The *Coordinator* is then linked to the bending `SingleProcess` and executed only when a new MU enters in that node of the pro-

Table 3.9: Simulation times with different number of bending operations.

Operations	Abstracted Models		
	Simplified	Detailed	Simulated Time
1	0.016	0.031	1.22
10	0.176	0.318	8.70
20	0.384	0.549	16.70
50	0.928	1.307	39.28

duction line. The simulation resolution is MU-Accurate, meaning that the switching actions can be performed only at the entrance of an MU in the `SingleProcess`. When the execution of the bending operation is completed, the *Coordinator* retrieves the executed time to bend the metal sheet and returns it to the `SingleProcess` of Plant Simulation.

The simulation results for the proposed approach are reported in Table 3.9. In the table are detailed the execution time to simulate the metal sheet bending with both *simplified* and *detailed* model, as well as the simulated time. The approach is able to efficiently integrate the CPSs inside the production line simulator, with a simulation time that is sensibly lower than the simulated time. The presented approach is an "enabling technique" allowing the integration of complex CPS models into a production line which allows adding more layers of information, like machine wearing, aging, and maintenance of this machineries.

3.10 Conclusions

The methodology presented in this chapter is the first step towards the holistic simulation of an entire CPS. We have seen the process of abstraction and integration through the mixed-signal scheduling approach.

The *abstraction* methodology removes all the details meaningless whenever to simulate only the functionality of an entire platform. It preserves only the relations among a subset of the physical quantities of the components, that represent inputs and outputs of interest of the component within the platform. Furthermore, it can deal with analog descriptions belonging to different physical domains like *electric*, *rotational*, *magnetic*, *kinematic*, *etc*.

The *mixed-signal scheduling* methodology integrates analog and digital processes and deals with cross-domain analog functions. Automatic code generation allows generating a C++ virtual prototype of mixed-signal devices to simulate an entire heterogeneous virtual platform without introducing computational overhead due to synchronization of multiple tools as required by current mixed-signal simulators.

The abstraction from *transistor-level to behavioral-level* shows how the non-linear designs can be replaced by state-dependent small-signal models. This is a key step which allows applying the methodology of abstraction to functional-level to a broader set of industrial designs.

The *waveform relaxation interfaces* allow overcoming the limitations of the analog abstraction methodology. The improve the scalability of the approach by splitting the conservative nets and reducing the number of symbols in each system of equations. Furthermore, they allow composing abstracted models, drastically reducing the time required to perform the abstraction, which needs to be done only once for each macro-block.

The integration of the holistic platform inside the production line of a factory opens the way to new studies for Industry 4.0 and smart manufacturing. Especially new techniques could be developed to improve the predictive maintenance of factory machineries.

Finally, the flow proposed in this chapter has been implemented in the automatic tool ASTRAL, and tested on a set of different configurations of a smart system heterogeneous platform. The results explores three aspects of the proposed methodology: 1) the ability to deal with multi-domain components, 2) the efficiency when dealing with realistic multi-domain devices, 3) the synchronization overhead reduction achieved with the mixed-signal scheduling, and 4) the potentials of the transitor to behavioral-level abstraction.

4

Network synthesis for cyber-physical systems

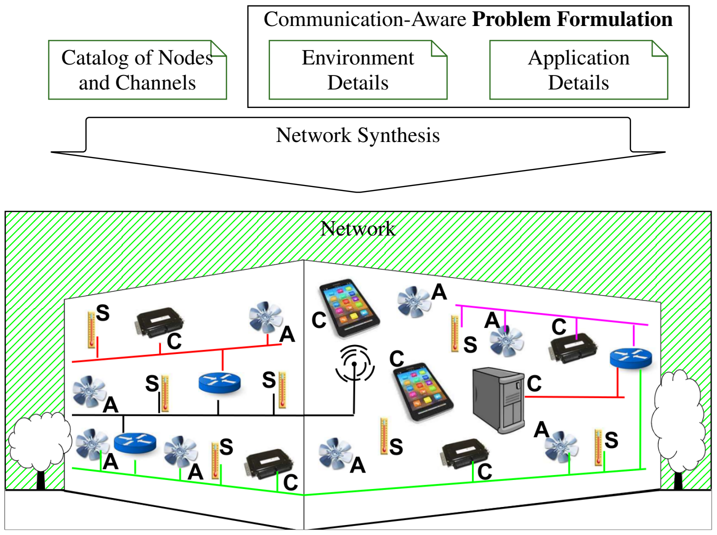

Communication-Aware Problem Formulation
Catalog of Nodes and Channels
Environment Details
Application Details
Network Synthesis
Network
S
C
A

Fig. 4.1: Overview of the network synthesis flow, used to design a networked cyber-physical system for building automation.

The previous chapter introduces a methodology for the holistic simulation of a Cyber-Physical System (CPS), comprising a series of digital and analog components to perceive/react/manipulate the outside world. It is clear that the target scope of the approach until now is that of the platform and at most the surrounding environment. However, recent advances in communications technologies open to entirely new distributed applications in which hundreds or thousands of CPSs interact together through different types of channels and protocols [56, 57, 58]. We can define them as Networked Cyber-Physical Systems (NCPSs) since these applications present three critical features that distinguish them from traditional network-connected applications, *i.e.*, *1)* the communication aspects affect the design flow of the CPSs, *2)* system-of-systems nature, and *3)* strict relation with the surrounding physical environment.

To clarify the discussion, let us introduce an example related to the temperature control of a building as depicted at the bottom of Figure 4.1. Several sensors (in figure denoted by *S*) detect local temperature. Collected data are sent to controllers (denoted by *C*), which send commands to actuators (denoted by *A*), *e.g.*, coolers. Controllers decide the activation of actuators according to various policies both centralized and distributed; for example, a controller may be present in each room to adjust the local temperature, but a centralized controller is also present to ensure that room settings comply with the total energy budget. A controller application can also be executed by personal mobile devices so that each user can control the temperature of the currently occupied space according to a personal profile.

In NCPSs, the *communication aspects* affect the design flow. Considering Figure 4.1, physical channels among nodes can be either wireless or wired according to deployment constraints (*e.g.*, cabling costs and feasibility in historical buildings) and mobility requirements. Communication protocols depend on the type of these physical channels, on required reliability and quality of service (*e.g.*, maximum latency). Assuming that highly optimized nodes are desirable, the choice of physical channels affects the definition of hardware network interfaces while the choice of communication protocols affects the memory and computational requirements of the hardware platform. All these aspects are formalized in the *Communication-aware Problem Formulation*, which is the starting point for the design flow for NCPSs proposed in this work.

Up to now, the CPSs design flow has jointly addressed hardware and software aspects, while communication aspects have been faced separately by a different research community. This lack of coordination may lead to non-optimal solutions in the system design; past work demonstrated that hardware/software design and network design are correlated [59]. To further push performance, energy saving and reliability, the network among nodes should be jointly designed with hardware and software components [60]. In particular, Computer-Aided Design (CAD) should be fruitfully applied not only to each node, as currently done in the context of electronic systems design but also to the network among them. For this reason, a *Communication-aware Design Flow* is required.

A NCPS can be seen as a *System-of-Systems* since even if the various nodes can independently operate, they interact together to achieve the *good behavior of the global application* [61]. In the mentioned example, the final objective is to achieve a good control of the temperature, and it does not matter the set of nodes that provides such functionality, as long as the global application behavior satisfies design objectives. Thus, NCPSs pose new questions to designers, traditionally mainly interested in the specification of each single network node as done for Internet servers and clients. Most relevant issues are:

- finding the optimal number of nodes to achieve the common mission;

- finding the best assignment (according to given metrics) between software tasks and hosting nodes by taking into account tasks' requirements and nodes' capabilities;
- finding the best set (according to given metrics) of network protocols by taking into account communication requirements and the presence of a legacy network infrastructure.

The last distinguishing feature is the *strict relationship with the environment*. Networked sensors and actuators should be placed where is required by the application. Furthermore, the environment affects communications in such systems; for instance, walls and distance may affect wireless communications whereas area size affects the deployment cost of wired infrastructure. Finally, the number and position of nodes affect the communications among them and application performance.

Solving these issues leads to the so-called *Network Synthesis*, *i.e.*, the allocation of functionality onto nodes and the complete definition of the communication infrastructure among them. The **contributions** proposed in this chapter are:

- a *communication-aware Design Flow* centered on Network Synthesis for NCPSs;
- a *communication-aware formal specification of the whole distributed system* to formulate Network Synthesis as an optimization problem;
- the formulation of Network Synthesis as a Mixed Integer Linear Programming (MILP) problem.

This work can be considered as part of the research roadmap started by [62] and continued by [63]. In [62] a preliminary version of the flow and the formal specification was described, but many issues were present, and the optimization strategy was missing. In [63] the focus was on model-driven design and simulation employing a UML-based representation. However, such representation does not allow, per se, the formulation of an optimization problem which was left in background.

The chapter is organized as follows. Related work is presented in Section 4.1. The building blocks of a formal representation for distributed embedded applications and the corresponding design flow are described in Section 4.2. The formulation of network synthesis as a MILP problem is provided in Section 4.3.1. The analysis of its complexity and scalability is reported in Section 4.4. Experimental results are provided in Section 4.5. Finally, conclusions and future work are reported in Section 4.6.

4.1 Related Work

The design of distributed systems relies on methodologies for their functional specification and techniques for network design.

4.1.1 System Functional Specification

The information driving network synthesis can be extracted from a platform-independent description of an application created using popular languages as those reported in this section.

The Modeling and Analysis of Real-Time and Embedded systems (MARTE) [64] is a Unified Modeling Language (UML) [65] profile designed to allow an easy specification of real-time and Cyber-Physical Systems (CPSs). It provides some subprofiles, like Non-Functional-Properties (NFPs), which allow describing the "fitness" of the system behavior (*e.g.*, performance, memory usage, energy consumption, *etc.*). The Software Resource Modeling (SRM) and the Hardware Resource Modeling (HRM) profiles are derived from NFP, and they address the modeling of resources. The System Modeling Language (SysML) [66] is an UML extension which provides a general-purpose modeling language for systems engineering applications.

Mathworks has developed Simulink [67] and Stateflow [68] to model and simulate dynamic and cyber-physical systems. The former allows representing an application as the inter-connection of analog or digital blocks while the latter allows describing applications as finite-state machines. They can also be combined representing hybrid automata.

The Ptolemy Project [69] was born to model concurrent real-time and CPSs. One of its main advantages is the support for heterogeneous mixtures of computation models. Ptolemy supports simulation by using the actor-oriented design; actors are software components executed concurrently and able to communicate by sending messages through interconnected ports. Ptolemy also supports communication modeling through Khan Process Networks (KPN), *i.e.*, groups of deterministic sequential processes which are communicating through unbounded FIFO channels [70].

They are distributed Models of Computation (MoCs) based on tokens which focus on the flow of computation; thus it seems well suited to check properties on the communication schema.

SystemC [71], initially born as a hardware description language, has been extended with the Transaction-Level Modeling (TLM) [72], to describe hardware/software systems. SystemC and TLM allow describing tasks as nested components with event-driven or clock-driven processes. Communications between tasks can be described by using standard protocols and payloads which simplify the specification of their behavior. TLM was born to represent local communications, such as bus interconnections or accesses to devices.

SpecC [73] is an extension of the C language to be used as a system-level design language, like SystemC/TLM. The SpecC methodology is a top-down design flow, with four distinct levels of abstraction. It provides different ways for describing the target control (sequential, FSM, parallel and behavioral). One key concept of SpecC is the clear separation of the communication and computation model which can be useful to specify computation and communication aspects of tasks.

Metropolis [74] is a framework based on the idea of a meta-model to support various communication and computation semantics consistently. This approach implements the abstract semantics of process networks and uses the concepts advocated by the Platform-Based Design (PBD) methodology, *i.e.*, functionality and architecture across models of computation and abstraction levels, and the mapping relationships between them.

4.1.2 Network Design

Network design has been addressed by many research works, in different fields, such as Wireless Sensor Networks (WSNs). In [75] a virtual architecture has been proposed in order to simplify the synthesis of WSNs algorithms. Network topology and high-level functionality are used to configure the virtual architecture. This work is mainly focused on the application part of the system rather than on communication aspects.

In [76], PBD has been adopted to design WSNs for industrial control. In PBD, the application is usually designed at a high level and then mapped onto a set of possible actual candidates for the nodes. However, no guideline is provided for selecting an appropriate network architecture and communication protocol. Scope-based techniques have been proposed in macro-programming to specify complex interactions between heterogeneous nodes of a WSN [77]. However, the nodes number and network topology are an input required by the technique and not a result, as in the proposed approach.

A tool for the optimal design of WSNs for building automation has been proposed in [78]. It suggests to integrate the network design flow with the knowledge about the routing algorithm used after the deployment of the network. Since the routing algorithm is known a priori, it further proposes to systematically introduce redundancy in order to maximize the performance of the chosen algorithm. Then, it proposes a sub-optimal polynomial-time heuristic for the synthesis problem and compares it with a custom formalization of the MILP proposed in [79].

A communication synthesis methodology and hardware/software integration for cyber-physical system design has been addressed in [80]. The method is based on task graphs and used after hardware/software partitioning and task scheduling. On one hand, this inversion has the advantage that the scheduling problem is simplified since communication components will be designed later. On the other hand, it does not consider that scheduling could be optimized if communication aspects were considered earlier.

The design of Network on Chips (NoCs) offers an example of computation-communication integrated approach which is close to the purpose of work. NoCs are CPSs which are designed with the traditional specification-refinement-synthesis flow; nevertheless, they have also a communication infrastructure which is a simplified version of a packet-switched network [81]. The design of the internal NoC communication infrastructure presents problems similar to the one of traditional packet-based networks [82]. For example, the design of NoC to meet hard latency constraints is addressed in [83]. The problem of the optimal mapping of tasks onto NoC's cores is known to be NP-hard. In some works, heuristics based on graph-decomposition techniques have been used [84, 85]. A MILP formulation of the problem has been proposed [86]. It assumes a regular 2D mesh topology and shortest-path static routing. This methodology allows two different optimization criteria, *i.e.*, minimization of the average hop distance (which is proportional to energy consumption and communication delay), as well as minimization of the bandwidth (which consists in minimizing the most-congested link-queuing time and maximizing the throughput). Network synthesis in NoCs is based on strong assumptions on network's features (*e.g.*, the topology); thus, such approaches are not general enough to be applied also to traditional networks as proposed in this work.

The efficient routing of communication paths gives another opportunity of optimization, also at different levels of the protocol stack. At the lowest level, a synthesis process for routing of physical wires inside an automotive system is proposed in [87]. It aims at meeting requirements about delay, quality of signal, power, and temperature. First, a Steiner tree is generated using a customized Kou-Markowsky-Berman (KMB) algorithm minimizing connections length. Then, a Linear Programming (LP) problem is formulated, and its solution is used to modify the Steiner tree such that the overall delay is minimized and signal quality maximized. At a higher level, Xu et al. [88] propose a MILP formulation applied to ZigBee wireless networks. It comprises four specific groups of constraints: devices placement, link activation for routing, connections scheduling and communication quality of service. Their formulation is limited to ZigBee architectures.

Synthesis of communication protocols is another research topic related to this work. Automatic tools have been adopted to derive the actual implementation of protocols specified through finite state machines [89, 90], Petri Nets [91], trace models [92], and languages like LOTOS [93]. All these approaches focus on the behavioral aspect of communication without taking into account the design of the nodes. A general modeling framework for a global design flow could be useful to allow the joint exploration of hardware, software and Network design space dimensions as addressed in the proposed approach.

4.2 Communication-aware Design Flow

The creation of a specific design flow for distributed CPSs requires the definition of new entities to formulate a design problem that accounts for communications; then, the traditional flow for cyber-physical systems can be extended to solve the problem. Both aspects are described in the following text.

4.2.1 Network Specification

This section introduces the entities and relationships representing the communication aspects to be designed in distributed CPSs. The proposed formal model is network-centric, *i.e.*, it describes the characteristics which are related to communications while all the other details are omitted or highly abstracted. The objective is the description of communication requirements as an optimization problem whose solution leads to the network synthesis. Therefore, this formalization is neither a distributed model of computation, as Kahn Process Networks, nor a language for executable specification as SystemC.

Figure 4.2 shows a general picture of such entities and their relationships. It consists of tasks (Section 4.2.1.a), implementing the behavior of the distributed system, which are hosted inside network nodes (Section 4.2.1.c). The stream of data

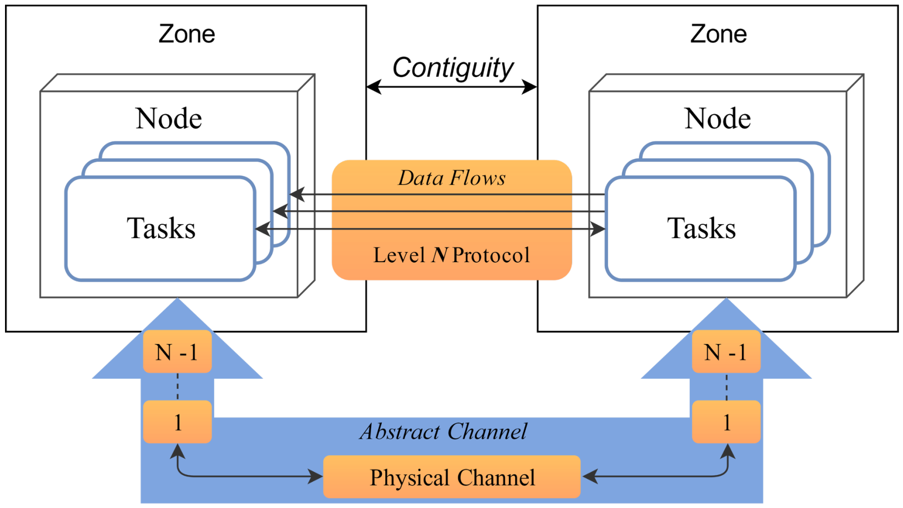


Fig. 4.2: Entities for the communication-aware specification.

between tasks is represented by data-flows (Section 4.2.1.b). Tasks and corresponding nodes are deployed inside specific partitions of the environment named zones (Section 4.2.1.e). Zones are related together by contiguity which models the influence of the environment on communications, *i.e.*, obstacles, walls, distances (Section 4.2.1.f).

An abstract channel (Section 4.2.1.d) is established between nodes to convey the data-flows of the hosted tasks. The intention is to generalize the concept of physical channel with an abstraction which takes into account also the presence of higher protocol layers (we refer to the ISO/OSI representation). The highest layer encompassed in the abstract channel depends on the type of protocols implemented in the conveyed data-flows. To understand the underlying idea of this generalization, let us consider two examples. In the first example, tasks implement a datalink protocol, *e.g.*, IEEE 802.11ac, and therefore the abstract channel represents the physical channel. In the second example, tasks implement a temperature control application through messages exchanged over a channel which is assumed to be reliable and byte-oriented. In this case, the abstract channel encompasses all the layers from physical channel up to TCP/IP.

In the following text, the network entities will be described in detail together with the relationships between them. Regarding notation, $\mathbb{R}_{\geq}$ is used to denote the non-negative real numbers, $\mathbb{R}_{[x,y]}$ identifies the real numbers between x and y, and $\mathbb{B} := \{true, false\}$ for boolean values. Furthermore, the term *Time Unit* (TU) refers to a timing value of one second and *Space Unit* (SU) to a distance of one meter.

4.2.1.a Tasks

A task represents a basic functionality of the whole application; it takes some data as input and provides some output. For network synthesis, the focus is not on the description of the functionality itself and its hardware/software implementation but rather on its computational and mobility requirements to decide its assignment to a given network node. A task $t = [s, m, z] \in \mathcal{T}$ is a triple defined as follows

$s \in \mathbb{R}_{\geq}$	represents the task size, *i.e.*, the computational resources required to perform its activity;
$m \in \mathbb{B}$	specifies whether the task should be placed on a mobile node;
$z \in \mathcal{Z}$	specifies to which zone the task belongs.

Defining the appropriate task size is a designer's responsibility. It would be easy to generalize the description to the case where $t.s \in \mathbb{R}^k_{\geq}$, allowing to consider an array of k different types of resources.

4.2.1.b Data-Flows

A data-flow (DF) represents the flow of messages between two tasks; output from the source task is delivered as input to the destination task. A data-flow $d = [st, dt, s, d, e] \in \mathcal{D}$ is characterized by the attributes

$st, dt \in \mathcal{T}$	are the source and destination tasks;
$s \in \mathbb{R}_{\geq}$	represents the data-flow size, *i.e.*, bit-rate;
$d \in \mathbb{R}_{\geq}$	indicates the maximum acceptable delay;
$e \in \mathbb{R}_{\geq}$	specifies the maximum acceptable error rate.

Network synthesis is mainly driven by the communication requirements of the data-flows which affect the choice of channels and protocols between the nodes hosting the involved tasks.

4.2.1.c Nodes

A node can be seen as a container of tasks. At the end of the whole design flow, nodes will be instances of hardware platforms with CPUs and network interfaces and tasks will be implemented as either custom hardware components or software processes. For network synthesis, the focus is on the resources made available by the node to host many tasks. A node $n = [s, k, e, te, ek, m] \in \mathcal{N}$ is a tuple whose attributes are as follows

$s \in \mathbb{R}_{\geq}$	represents the node size, *i.e.*, the available computational resources;
$k \in \mathbb{R}_{\geq}$	denotes the node economic cost;
$e \in \mathbb{R}_{\geq}$	is the intrinsic energy consumption of the node without considering the executed tasks;
$te \in \mathbb{R}_{\geq}$	determines the energy consumption of the tasks assigned to the node over a *TU* (each task t mapped into the node n consumes an amount of energy equal to $t.s$ times $n.te$);

$ek \in \mathbb{R}_{\geq}$	relates the consumed energy with a specific cost based on the energy source (*e.g.*, batteries, solar panels, energy service company, *etc.*);
$m \in \mathbb{B}$	identifies if the node is mobile or static.

The network synthesis process assigns tasks to nodes. Tasks with the mobile attribute set to true must be placed on mobile nodes.

Regarding energy consumption, there are two contributions. The first one, denoted by e, is constant and independent of task operations while the second, denoted by te, accounts for the energy consumed to execute each task operation. Regarding economic cost, there is a constant contribution, denoted by k, to acquire the use of the node (*e.g.*, because of purchase or rent) and a variable contribution due to energy consumption. To compute this contribution, we introduced ek which describes the cost of each energy unit. This cost depends on energy source, *e.g.*, cost of batteries and their replacement or energy service company bill. This unit cost can be zero in case of energy harvesting (*e.g.*, solar panels).

4.2.1.d Abstract Channels

An Abstract Channel (AC) can be seen as a container of data-flows. It is an ideal medium connecting two or more nodes. Referring to the ISO/OSI model, it is defined as follows:

Definition 1 (Abstract Channel). *Assuming that there is a data-flow implementing a level-N protocol, it is hosted by an AC which represents the physical channel and all the protocol entities up to level $N - 1$.*

An abstract channel $ac = [e, de, k, ek, w, pp, s, dl, er] \in \mathcal{A}$ is a tuple characterized as follows

$e \in \mathbb{R}_{\geq}$	is the intrinsic energy consumption of the channel without considering hosted data-flows;
$de \in \mathbb{R}_{\geq}$	is the energy required to send a bit through the channel over a *TU* (each data-flow d deployed inside the channel c consumes an amount of energy equal to $d.s$ times $c.de$);
$k \in \mathbb{R}_{\geq}$	specifies the economic cost of this communication architecture;
$ek \in \mathbb{R}_{\geq}$	relates the consumed energy with a specific cost based on the energy source;
$w \in \mathbb{B}$	specifies if the channel is wireless or wired;
$pp \in \mathbb{B}$	specifies if the channel is point-to-point;
$s \in \mathbb{R}_{\geq}$	specifies the channel size, *i.e.*, its capacity;
$dl \in \mathbb{R}_{\geq}$	specifies the maximum transmission delay of the channel;
$er \in \mathbb{R}_{\geq}$	specifies the maximum error rate of the channel.

Data-flows between mobile tasks (hosted by mobile nodes) can be assigned only to wireless abstract channels. The last three attributes of the AC represent the *Quality of Service (QoS)* resulting from the presence of a given physical channel and all encompassed protocols. Similar attributes are present in the data-flow description;

they represent the QoS *required* by the data-flow which should be *provided* by the hosting abstract channel. This is one of the driving rules of the network synthesis. Attribute $ac.pp$ distinguishes between point-to-point and multi-point channels. It is worth noting that this information is orthogonal to wireless/wired attribute. There are multi-point wired channels (*e.g.*, CAN bus) and point-to-point wireless channels (*e.g.*, Bluetooth connections and wireless bridges).

For economic cost and energy consumption, similar reasoning as for nodes applies since the proposed definition of Abstract Channel accounts for both the physical channel and possible intermediate systems (*e.g.*, switches and routers). Regarding energy consumption, there are two contributions. The first one, denoted by e, is constant and independent of communications while the second, denoted by de, accounts for the energy consumed to transmit data-flow bits. Regarding economic cost, there is a constant contribution, denoted by k, to acquire the use of the channel (*e.g.*, because of purchase or rent of the line and intermediate systems) and a variable contribution due to energy consumption. To compute this contribution, we introduced ek which describes the cost of each energy unit. This cost depends on energy source, *e.g.*, cost of batteries and their replacement or energy service company bill. This unit cost can be zero in case of energy harvesting (*e.g.*, solar panels).

4.2.1.e Zones

In distributed embedded applications tasks should be active in specific positions of the 3D space. In the mentioned example, temperature sensors and actuators are distributed in the various rooms of the building. Position of tasks is essential for their assignment to nodes. Then nodes position is also critical to determine the effect of obstacles and distance on communications between them. In general, we want to address properties like "between nodes n_i and n_j there is an obstacle." Information about precise 3D positioning may be not available and even not useful for a given application (for instance, a temperature sensor is required in each room, but its position may be not so relevant). Therefore we propose to describe the position of tasks and nodes in the 3D space by partitioning it according to application needs (*e.g.*, rooms) and the presence of communication-relevant properties such as obstacles and distances. We denote by $\mathcal{Z}$ the set of Zones generated by this partition.

4.2.1.f Contiguity Relationship

Contiguity relationship describes the relationship between zones from the communication perspective. We assume that nodes placed in the same zone are always able to communicate with the default quality of service of the involved abstract channel (see Section 4.2.1.d). If nodes are deployed into different zones, the quality of service might drop because of distance or obstacles. The level of degradation also depends on the type of abstract channel. Furthermore, in case of wired channels, the relationship between zones can be also used to capture the wiring cost. A contiguity element $cnt = [z_1, z_2, ac, c, dc] \in \mathcal{C}$ is a tuple whose attributes are characterized as follows

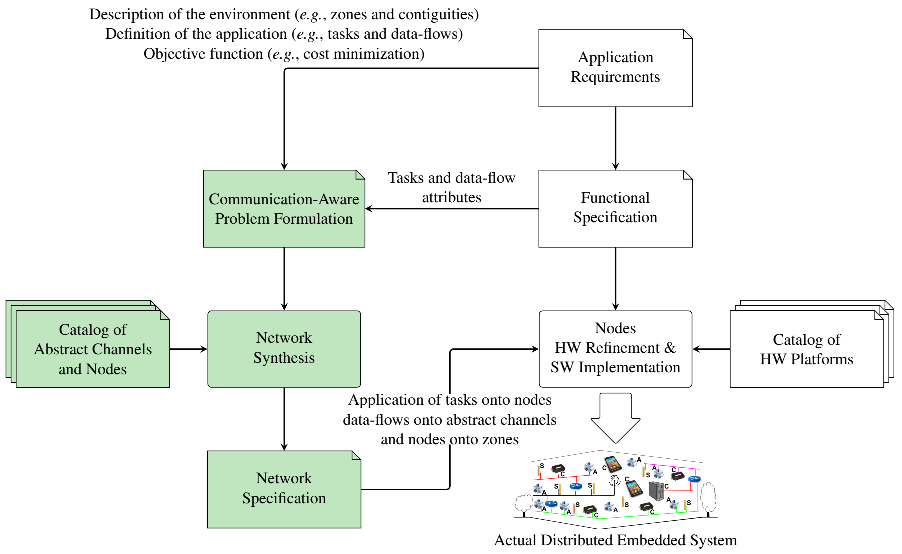

Description of the environment (*e.g.*, zones and contiguities)
Definition of the application (*e.g.*, tasks and data-flows)
Objective function (*e.g.*, cost minimization)
Application Requirements
Communication-Aware Problem Formulation
Tasks and data-flow attributes
Functional Specification
Catalog of Abstract Channels and Nodes
Network Synthesis
Nodes HW Refinement & SW Implementation
Catalog of HW Platforms
Application of tasks onto nodes
data-flows onto abstract channels
and nodes onto zones
Network Specification
Actual Distributed Embedded System

Fig. 4.3: Proposed design flow for networked cyber-physical systems: the new steps for network design (in light green on the left) are added symmetrically to the state-of-art design flow (in white on the right).

$z_1, z_2 \in \mathcal{Z}$	are the involved zones;
$ac \in \mathcal{A}$	is the abstract channel to which the contiguity applies;
$c \in \mathbb{R}_{[0,1]}$	is the attenuation coefficient to compute the remaining level of QoS of the given abstract channel ac after crossing the border between the given zones;
$dc \in \mathbb{R}_{\geq}$	represents the wiring cost to deploy the given channel between the given pair of zones; this attribute is relevant only for wired channels moreover, takes into account both medium type and length.

4.2.2 Design Flow

White boxes in the right part of Figure 4.3 represent the traditional design flow of CPSs. The starting point is the set of *Application Requirements* both functional and non-functional. A platform-independent *Functional Specification* is created starting from application requirements. Interacting components are expressed through languages like UML and C/C++ or through the use of tools like Matlab/Simulink/Stateflow (see Section 4.1.1). Concerning the entities defined in Section 4.2.1, a functional specification can be given as a set of Tasks exchanging information through Data-Flows.

This specification, together with a description of the target platform, is the subject of a Design Space Exploration (DSE) which maps Tasks onto hardware and software components of the target platform. The result is a platform-dependent description of the system, in which the hardware blocks correspond to actual devices while the software is implemented and compiled for the target processors. Such flow is well suited for isolated CPSs. However, in case of distributed applications made of many CPSs, it lacks a specific path devoted to the design of the communication infrastructure among them.

For this reason, this work proposes to extend the flow with new steps shown in light green on the left side. The new design path is quite symmetric *w.r.t.* the traditional one since it applies the same concepts to the communication aspects of the whole system. A *Communication-aware Problem Formulation* for the whole application is created by using information taken from the *Application Requirements* and the *Functional Specification*. Such information can be described concerning the entities defined in Section 4.2.1.

The *Application Requirements* block provides:

- a description of the environment as a set of Zones and Contiguity relationships among them;
- a definition of the application as a set of Tasks and Data-Flows with Task-Zone assignments;
- an optimization objective function (*e.g.*, energy minimization).

The *Functional Specification* allows to obtain the attributes of Tasks and Data-Flows. Data-Flow attributes represent communication constraints of the various data-flows of the distributed application, *e.g.*, their bit-rate as well as maximum acceptable delay and error rate.

The *Communication-aware Problem Formulation* describes a constrained optimization problem which links metrics to be optimized with constraints to be satisfied. This problem description, together with a description of available abstract channels and nodes (defined in Section 4.2.1), is the subject of DSE aiming at searching the optimal solutions. Similarly to how components are defined in electronic system design, this process is named *Network Synthesis* and is defined as follows:

Definition 2 (Network Synthesis). *Network synthesis is a design process which starts from an optimization problem and finds a feasible solution which defines its communication infrastructure regarding mapping of application Tasks onto network Nodes, their spatial displacement onto Zones, the type of channels and protocols among them, and the network topology.*

The final result is the *Network Specification* which contains critical information for the design of each node of the network, *i.e.*, the list of functions assigned to it and the presence of new computation tasks to handle network protocols. For this reason, this description is used as input in the traditional DSE of each node as reported in the right part of Figure 4.3. The proposed flow has the following advantages which match with the properties of NCPSs:

- Network features are decided before the design of hardware and software components; in this way, the impact of communications can be taken into account in the early phase of the design process.

- The environment is taken into account during network design, and therefore its impact is considered in the following design of hardware and software components.
- The proposed top-down approach for the design of the distributed CPS matches with its nature of system-of-systems. In the context of network deployment, the traditional approach is bottom-up by making some implementation-specific assumptions based on designer's experience. For instance, the designer starts assuming to build an Ethernet network, and then connects Ethernet nodes and switches without considering if other technologies may be more suitable. To the best of our knowledge, this is the first proposal that considers all available network architectures at the beginning of the flow.
- The decomposition of the application functionality into tasks and their allocation to nodes allows to distribute a single massive function over multiple nodes and the process is driven by the optimization objective, *i.e.*, cost, reliability, and so on.

4.3 Network Synthesis

The network synthesis problem is the core of the previous design flow. It can be formulated as an optimization problem by using the entities defined in Section 4.2.1. Among several optimization techniques that can be used to solve this problem, a mixed-integer linear problem (MILP) is presented and solved.

4.3.1 Problem Formulation

Referring to Figure 4.3 and using the entities defined in Section 4.2.1 it is now possible to formulate the *network synthesis* problem.

The Application Requirements allow to obtain the set of tasks (denoted by $\mathcal{T}$), data-flows ($\mathcal{D}$), zones ($\mathcal{Z}$) and contiguity elements ($\mathcal{C}$). The Functional Specification allows to obtain the attributes of tasks and data-flows. All these information elements allow to build the *Communication-aware Problem Formulation*.

Network synthesis process is also fed by the set of nodes (denoted by $\mathcal{N}$) and abstract channels ($\mathcal{A}$) which represent the technological libraries for this design space exploration. For each type of node and abstract channel, its name and attributes are specified.

Network synthesis can be formulated as an optimization problem in which the allocation of tasks onto nodes and data-flows onto abstract channels is driven by a set of constraints and metrics to be optimized. A possible way to formulate and solve such problem consists in describing it as a MILP problem. Independently of the formulation technique, the following strong constraints should be always considered:

- a non-mobile node cannot host a mobile task;
- a task with a given computational requirement cannot be hosted by a node which does not provide at least such resources;

- a data-flow with a given QoS requirement cannot be hosted by an abstract channel which does not provide at least such QoS;
- abstract channel types cannot be used between zone pairs whose contiguity brings to zero their QoS.

To model these general constraints, the following functions are defined and populated during a preprocessing phase:

- $\alpha_n(t)$, $t \in \mathcal{T}$ returns the set of *allowed nodes* to which the task t can be mapped;
- $\alpha_t(n)$, $n \in \mathcal{N}$ returns the set of *allowed tasks* which can be mapped into a node of type n;
- $\alpha_c(d)$, $d \in \mathcal{D}$ returns the set of *allowed channels* in which the data-flow d can be mapped. It can be further subdivided into $\alpha_{wc}(d)$ and $\alpha_{cc}(d)$ that only considers respectively the allowed *wireless* and *wired* channels;
- $\alpha_c(z_1, z_2)$, $z_1, z_2 \in \mathcal{Z}$, $z_1 \neq z_2$ returns the set of *allowed channels* which can be used to connect two nodes deployed respectively in z_1 and z_2. Two subsets $\alpha_{wc}(z_1, z_2)$ and $\alpha_{cc}(z_1, z_2)$ are defined, respectively identifying the allowed *wireless* and *wired* channels;
- $\alpha_d(c)$, $c \in \mathcal{A}$ returns the set of *allowed data-flows* which can be mapped into a channel of type c;
- $cont(z_1, z_2, ac)$, $z_1, z_2 \in \mathcal{Z}$, $ac \in \mathcal{A}$, is a hash function that allows to efficiently retrieve the contiguity relationship and thus the corresponding conductivity c and the wiring cost dc.

The tasks connected to a data-flow and the zones in which they are placed is well-known from Application Requirements. Therefore, given an abstract channel c, the set $\alpha_d(c)$ does not contain data-flows whose tasks are placed in zones across which c has zero conductivity.

4.3.2 MILP Variables

In the following, all the variables used during the MILP formalization are presented and explained in detail. The first two sets of variables play a distinguished structural role, in that they imply the space of all the other variables.

- $N_{n,z}$, $n \in \mathcal{N}$, $z \in \mathcal{Z}$ for each node-type $n \in \mathcal{N}$ and zone $z \in \mathcal{Z}$, $N_{n,z}$ denotes how many nodes of node-type n are deployed in zone z.
- C_c, $c \in \mathcal{A}$ for each channel type $c \in \mathcal{A}$, C_c states how many channels of type c are activated by the solution.

Since we can not write MILPs with an infinite number of variables, we need to rely on the following two parameters which can be conveniently computed in a preprocessing phase.

- $\overline{N}_{n,z}$, $n \in \mathcal{N}$, $z \in \mathcal{Z}$ for each node-type $n \in \mathcal{N}$ and zone $z \in \mathcal{Z}$, parameter $\overline{N}_{n,z}$ provides an upper bound on the value of $N_{n,z}$. Before running our model, we need to fix parameter $N_{n,z}$ to a natural value. We want this value to be as small as possible since the number of variables allocated by our MILP grows polynomially in $N_{n,z}$. However, we should make sure that there exist optimal solutions in which $N_{n,z} \leq \overline{N}_{n,z}$, *i.e.*, $\overline{N}_{n,z}$ should be a valid upper bound.

- $\overline{C}_c$, $c \in \mathcal{A}$ for each channel type $c \in \mathcal{A}$, parameter $\overline{C}_c$ provides an upper bound on the value of C_c. This means that, as above, we should make sure that there exist optimal solutions in which $C_c \leq \overline{C}_c$, or that we are ready to anyhow limit our search for good solutions below this parameter. Again, we want $\overline{C}_c$ to be as small as possible since the number of variables allocated by our MILP grows polynomially in this parameter.

This work proposes to consider

$$\overline{N}_{n,z} := |\{t \in \alpha_t(n) | t.z = z\}| \tag{4.1}$$

$$\overline{C}_c := |\{d \in \alpha_d(c)\}| \tag{4.2}$$

Bound 4.1 indeed provides an upper bound to the number of type-node n in zone z based on the number of allowed tasks for node n in zone z. Bound 4.2 is also a valid upper bound since the upper-bound of a given channel c is equal to the number of allowed data-flow inside that channel.

The purpose of our first set of boolean variable x is to activate, in a well structured way, single instances of nodes of any given type. For each node $n \in \mathcal{N}$, $z \in \mathcal{Z}$ and $p \leq \overline{N}_{n,z}$, their intended value is as follows

$$x_{n,z,p} = \begin{cases} 1 & \text{if there are at least } p \text{ nodes of type } n \text{ allocated in zone } z, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall n \in \mathcal{N}, \quad \forall z \in \mathcal{Z}, \quad \forall p \leq \overline{N}_{n,z} \tag{4.3}$$

Analogously, the second set of variables y determines the number of allocated channels of any given type

$$y_{c,p} = \begin{cases} 1 & \text{if at least } p \text{ channels of type } c \text{ are allocated,} \\ 0 & \text{otherwise.} \end{cases} \qquad \forall c \in \mathcal{A}, \quad \forall p \leq \overline{C}_c \tag{4.4}$$

Another issue is represented by the presence of point-to-point channels, which can only be deployed between two nodes. Such aspect has been formalized with the support of two variables. First, the tasks of a data-flow and a third task are related by a variable γ, whose formalization follows

$$\gamma_{d,t} = \begin{cases} 1 & \text{if tasks } d.st,\ d.dt, \text{ and } t \text{ are mapped into 3 different nodes,} \\ 0 & \text{otherwise.} \end{cases} \qquad \forall d \in \mathcal{D}, \quad \forall t \in \mathcal{T}, \tag{4.5}$$

Variable ρ instead, is formalized as

$$\rho_{t_1,t_2} = \begin{cases} 1 & \text{if tasks } t_1 \text{ and } t_2 \text{ are mapped into different nodes,} \\ 0 & \text{otherwise.} \end{cases} \tag{4.6}$$
$$\forall t_1, t_2 \in \mathcal{T}, \quad t_1 \neq t_2$$

The key aspects of the proposed formulation are the assignment of tasks to nodes and the deployment of data-flows into channels. Concerning the positioning of tasks inside nodes, a new boolean variable $w_{t,n,p}$ defined as

$$w_{t,n,p} = \begin{cases} 1 & \text{if task } t \text{ is associated with the } p\text{-th node of type } n \text{ in zone } t.z, \\ 0 & \text{otherwise.} \end{cases} \tag{4.7}$$
$$\forall t \in \mathcal{T}, \quad \forall n \in \alpha_n(t), \quad \forall p \leq \overline{N}_{n,z}$$

Deployment of data-flows inside channels is identified by a variable $h_{d,c,p}$ defined as

$$h_{d,c,p} = \begin{cases} 1 & \text{if the data-flow } d \text{ is placed in the } p\text{-th channel of type } c, \\ 0 & \text{otherwise.} \end{cases} \tag{4.8}$$
$$\forall d \in \mathcal{D}, \quad \forall c \in \alpha_c(d), \quad \forall p \leq \overline{C}_c$$

Variable q_{c,z_1,z_2} is statically solved before executing the optimization. It is initialized by checking if the conductance of the channel between the two zones is greater than zero. It is defined as follows

$$q_{c,z_1,z_2} = \begin{cases} 1 & \text{if a channel of type } c \text{ can connect nodes inside } z_1 \text{ and } z_2, \\ 0 & \text{otherwise.} \end{cases} \tag{4.9}$$
$$\forall z_1, z_2 \in \mathcal{Z}, \forall c \in \mathcal{A}$$

Finally, variable $j_{c,p}$ is introduced in order to keep track of the deployment cost for each instance of deployed channel. It is defined as

$$j_{c,p} \in \mathbb{R}_{\geq} \qquad \forall c \in \mathcal{A}, \forall p \leq \overline{C}_c \tag{4.10}$$

4.3.3 MILP Objectives

Four metrics were considered to be of significant importance within a distributed CPS, and then subject to optimization. These metrics are: *Economic cost*, *Energy consumption*, *Transmission delay*, and *Error rate*. For all the metrics described above the optimization can be determined by a minimization function.

4.3.3.a Economic Cost Minimization

Its objective is to minimize the total economic cost of the distributed CPS, and it is defined as follows

$$\min \begin{bmatrix} \sum_{n}^{\mathcal{N}} \sum_{z}^{\mathcal{Z}} \sum_{p=1}^{\overline{N}_{n,z}} (x_{n,z,p} * (n.k + n.e * n.ek)) + \\ \sum_{c}^{\mathcal{A}} \sum_{p=1}^{\overline{C}_c} (y_{c,p} * (c.k + c.e * c.ek + j_{c,p})) + \\ \sum_{t}^{\mathcal{T}} \sum_{n}^{\alpha_n(t)} \sum_{p=1}^{\overline{N}_{n,t.z}} (w_{t,n,p} * n.te * t.s * n.ek) + \\ \sum_{d}^{\mathcal{D}} \sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} \frac{h_{d,c,p} * c.de * c.ek * d.s}{cont(d.st.z, d.dt.z, c).c} \end{bmatrix} \tag{4.11}$$

The first two sums of the metric considers the base cost of deployed nodes and channels plus their energetic cost. The therm j_c, p inside the second sum consider the supplementary cost for wired channels whenever they are placed between different zones. The last two sums concerns the economic cost deriving from the consumed energy. For tasks, this is done by multiplying the total energy consumed by a task deployed inside a node for the specific energy cost for that node. Similarly, the last sum considers the energetic cost of data-flows by multiplying the total amount of energy consumed by a deployed channel for the price of the energy for that particular channel.

4.3.3.b Energy Consumption Minimization

The second optimization objective is to minimize the total energy consumption of the distributed CPS, and is defined as follows

$$\min \begin{bmatrix} \sum_{n}^{\mathcal{N}} \sum_{z}^{\mathcal{Z}} \sum_{p=1}^{\overline{N}_{n,z}} (x_{n,z,p} * n.e) + \\ \sum_{c}^{\mathcal{A}} \sum_{p=1}^{\overline{C}_c} (y_{c,p} * c.e) + \\ \sum_{t}^{\mathcal{T}} \sum_{n}^{\alpha_n(t)} \sum_{p=1}^{\overline{N}_{n,t.z}} (w_{t,n,p} * n.te * t.s) + \\ \sum_{d}^{\mathcal{D}} \sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} \frac{h_{d,c,p} * c.de * d.s}{cont(d.st.z, d.dt.z, c).c} \end{bmatrix} \tag{4.12}$$

The first two sums of the metric consider the energy consumed by deployed nodes and channels. The third sum takes into account the task's resource requirements and multiplies it for the coefficient used to calculate contribution of each task to the node energy consumption (*i.e.*, attribute $n.te$). The last sum multiplies the size of dataflows for the contribution to the energy consumption of channels where they are deployed (*i.e.*, attribute $c.de$).

4.3.3.c Transmission Delay Minimization

Its purpose is to minimize the total transmission delay of the distributed CPS. Follows its definition

$$\min \left[\sum_{d}^{\mathcal{D}} \sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} \frac{h_{d,c,p} * c.dl}{cont(d.st.z, d.dt.z, c).c} \right] \tag{4.13}$$

The above metric sums the transmission delay of channels where dataflows are deployed, enhanced by the effects of the border between the involved zones on the communication quality. This metric considers the delay for each dataflow and not only once for each deployed channel.

4.3.3.d Error Rate Minimization

The optimization objective is to minimize the total error rate of the distributed CPS. The function has the same structure as for the transmission delay minimization but, instead of summing the channel delay, its error rate value is used. Follows its definition

$$\min \left[\sum_{d}^{\mathcal{D}} \sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} \frac{h_{d,c,p} * c.er}{cont(d.st.z, d.dt.z, c).c} \right] \tag{4.14}$$

4.3.4 MILP Constraints

Constraints on the Number of Instantiated Components

The first group of constraints activates in accordance to the number of nodes and channels as well as defining the values of the upper-bounds of such components. More in details constraints C.1 and C.2 concerns the nodes. For all $n \in \mathcal{N}$ and $z \in \mathcal{Z}$

$$\begin{gathered} N_{n,z} = \sum_{p=1}^{\overline{N}_{n,z}} x_{n,z,p} \\ \forall n \in \mathcal{N},\ \forall z \in \mathcal{Z} \end{gathered} \tag{C.1}$$

$$\begin{gathered} N_{n,z} \geq p * x_{n,z,p} \\ \forall n \in \mathcal{N},\ \forall z \in \mathcal{Z},\ \forall p \leq \overline{N}_{n,z} \end{gathered} \tag{C.2}$$

Moreover, the second set of constraints C.3 and C.4, concerns the channels.

$$C_c = \sum_{p=1}^{\overline{C}_c} y_{c,p} \qquad \forall c \in \mathcal{A} \tag{C.3}$$

$$C_c \geq p * y_{c,p} \qquad \forall c \in \mathcal{A},\ \forall p \leq \overline{C}_c \tag{C.4}$$

Constraints on the Existence of Used Components

Constraints C.5 and C.6 ensure that nodes and channels are instantiated whenever tasks and data-flows use them.

$$w_{t,n,p} \leq x_{n,t.z,p} \qquad \forall t \in \mathcal{T},\ \forall n \in \alpha_n(t),\ \forall p \leq \overline{N}_{n,t.z} \tag{C.5}$$

$$h_{d,c,p} \leq y_{c,p} \qquad \forall d \in \mathcal{D},\ \forall c \in \alpha_c(d),\ \forall p \leq \overline{C}_c \tag{C.6}$$

Constraints C.7 and C.8 instead ensure that only the nodes and channels which are necessary are activated.

$$x_{n,z,p} \leq \sum_{t}^{(\alpha_t(n) \wedge t.z=z)} w_{t,n,p} \qquad \forall n \in \mathcal{N},\ \forall z \in \mathcal{Z},\ \forall p \leq \overline{N}_{n,z} \tag{C.7}$$

$$y_{c,p} \leq \sum_{d}^{\alpha_d(c)} h_{d,c,p} \qquad \forall c \in \mathcal{A},\ \forall p \leq \overline{C}_c \tag{C.8}$$

Constraints on Components Capacity

The assignment of tasks to nodes has to be compliant with the size (*i.e.*, resources) of each involved node. Constraint C.9, ensures that the total amount of resources required by tasks inside a given node is at most the size of the node.

$$\sum_{t}^{(\alpha_t(n) \wedge t.z=z)} t.s * w_{t,n,p} \leq n.s \qquad \forall n \in \mathcal{N},\ \forall z \in \mathcal{Z},\ \forall p \leq \overline{N}_{n,z} \tag{C.9}$$

Constraint C.10, ensures that the total amount of bit-rate used by data-flows mapped into a given channel is at most the size (*i.e.*, capacity) of the channel. Furthermore, the effect of the environment has to be taken into consideration.

$$\sum_{d}^{\alpha_d(c)} \frac{d.s * h_{d,c,p}}{cont(d.st.z, d.dt.z, c).c} \leq c.s \qquad \forall c \in \mathcal{A},\ \forall p \leq \overline{C}_c \tag{C.10}$$

Constraints on Tasks and Data-Flows Assignment

Tasks and data-flows are unique entities, specific of the application functionality; thus, they must be assigned only once to nodes and channels, respectively. For what concerns tasks, Constraint C.11, ensures that they are assigned to a node only once.

$$\begin{gathered}\sum_{n}^{\alpha_n(t)} \sum_{p=1}^{\overline{N}_{n,t.z}} w_{t,n,p} = 1 \\ \forall t \in \mathcal{T}\end{gathered} \tag{C.11}$$

However, for what concerns data-flows, their placement depends on whether the tasks which they connect reside in the same node or not. In the former case, formalized in Constraint C.12, the data-flow is not necessarily assigned to a channel, and its placement depends on variable ρ. For data-flows which instead have tasks which reside in different zones their placement inside a channel is necessary and ensured by Constraint C.13.

$$\begin{gathered}\sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} h_{d,c,p} = \rho_{d.st,d.dt} \\ \forall d \in \mathcal{D},\ d.st.z = d.dt.z\end{gathered} \tag{C.12}$$

$$\begin{gathered}\sum_{c}^{\alpha_c(d)} \sum_{p=1}^{\overline{C}_c} h_{d,c,p} = 1 \\ \forall d \in \mathcal{D},\ d.st.z \neq d.dt.z\end{gathered} \tag{C.13}$$

Constraints on Point-to-Point Channels

The next sets of constraints concern point-to-point channels, which have to abide a more tightening rule. Each of them can connect no more than a pair of nodes. Constraint C.14 ensures that variable ρ is correctly set whenever two tasks are mapped into different nodes. Constraint C.15 instead, sets ρ to constant 1 when the pair of tasks resides in different zones.

$$\begin{gathered}\rho_{t,t'} \geq (w_{t,n,p} + w_{t',n',p'} - 1) \\ \begin{array}{lll} \forall t, t' \in \mathcal{T}, & t.z = t'.z, & t \neq t', \\ \forall n \in \alpha_n(t), & \forall p \in \overline{N}_{n,t.z}, & \\ \forall n' \in \alpha_n(t'), & \forall p' \in \overline{N}_{n',t'.z}, & \\ (n \neq n') \vee (p \neq p') & & \end{array}\end{gathered} \tag{C.14}$$

$$\begin{gathered}\rho_{t,t'} = 1 \\ \forall t, t' \in \mathcal{T},\ t.z \neq t'.z, t \neq t'\end{gathered} \tag{C.15}$$

Constraint C.16 has to keep track of all the data-flows which have a node in common. This is necessary since whenever a data-flow mapped into a point-to-point channel share a node with another data-flow, the source and destination task of the latter have to be mapped into the same node *w.r.t.* the tasks of the former.

$$\begin{gathered}
\gamma_{d,d'.st} \leq 2 - h_{d,c,p} - h_{d',c,p} \\
\text{with } (d.st \neq d'.st) \wedge (d.dt \neq d'.st) \\
\gamma_{d,d'.dt} \leq 2 - h_{d,c,p} - h_{d',c,p} \\
\text{with } (d.st \neq d'.dt) \wedge (d.dt \neq d'.dt) \\
\forall c \in \mathcal{A},\ \forall p \leq \overline{C}_c,\ \forall d, d' \in \alpha_d(c), \\
c.pp = true, d \neq d'
\end{gathered} \tag{C.16}$$

Constraint C.17 ensures that, if the tasks of a data-flow d and a third task t are placed all in different nodes (C.14, C.15 and C.17), then data-flow d and the one connected to task t must be mapped into different channels (C.16).

$$\begin{gathered}
\gamma_{d,t} = 0 \\
\text{with } (d.st = t) \vee (d.dt = t) \\
\gamma_{d,t} = 1 \\
\text{with } (d.st.z \neq t.z) \wedge (d.dt.z \neq t.z) \wedge (d.st.z \neq d.dt.z) \\
\gamma_{d,t} \geq \rho_{t,d.st} + \rho_{t,d.dt} + \rho_{d.st,d.dt} - 2 \\
\text{with } (d.st.z = t.z) \vee (d.dt.z = t.z) \vee (d.st.z = d.dt.z) \\
\forall d \in \mathcal{D},\ \forall t \in \mathcal{T}
\end{gathered} \tag{C.17}$$

Constraints on Wireless Channels

Constraint C.18 ensures that whenever two data-flows are placed inside the same wireless channel, all tasks of the data-flows can communicate with each other. Thus, the conductivity with the given channel between the four combinations of the zones in which the tasks reside is greater than zero.

$$\begin{aligned}
h_{d,c,p} + h_{d',c,p} \leq 1 + (&q_{c,d.st.z,d'.st.z} * \\
&q_{c,d.st.z,d'.dt.z} * \\
&q_{c,d.dt.z,d'.st.z} * \\
&q_{c,d.dt.z,d'.dt.z}) \\
\forall c \in \mathcal{A},\ c.w = true,\ \forall p \leq \overline{C}_c,\ &\forall d, d' \in \alpha_d(c),\ d \neq d'
\end{aligned} \tag{C.18}$$

Constraints on deployment cost of wired channels

Finally, Constraint C.19 poses an lower-bound on variable $j \in \mathbb{R}_{\geq}$. Such lower-bound is equal to the *highest* deployment cost for those channels which are placed between two zones. It can be appreciated that such constraint is defined only for those pairs of zone interested by data-flows. The upper-bound on variable j is intrinsically ensured by the objective function, which aims at minimizing the variables.

$$j_{c,p} \geq h_{d,c,p} * cont(d.st.z, d.dt.z, c).dc$$

$$\forall c \in \mathcal{A},\ c.w = false,\ \forall p \leq \overline{C}_c, \qquad \text{(C.19)}$$
$$\forall d \in \alpha_d(c),\ d.st.z \neq d.dt.z$$

4.4 Complexity and Scalability

One should be aware that the network synthesis model proposed and solved in this chapter is strongly NP-hard even in the following two very extreme special cases:

- $|\mathcal{N}| = 1$, $|\mathcal{Z}| = |\mathcal{A}| = |\mathcal{D}| = 0$, $|\mathcal{T}| \in \mathbb{N}$. Without loss of generality, both size and cost of each node instance are 1. Every node instance can be regarded as a *bin* that we can open or not in order to accommodate a set of tasks, each one representing an *item* of the same size.
- $|\mathcal{A}| = |\mathcal{N}| = 1$, $|\mathcal{T}| = 2$, $|\mathcal{Z}| = 1$, $|\mathcal{D}| \in \mathbb{N}$. Without loss of generality, both size and cost of each channel instance are 1. Every channel instance can be regarded as a *bin* that we can open or not in order to accommodate a set of data-flows, each one representing an *item* of the same size.

As a consequence, unless P=NP, every general algorithm solving our model will exhibit a running time which is exponential *both* in $|\mathcal{T}|$ *and* in $|\mathcal{D}|$. Nonetheless, the MILP solution here offered works remarkably well in practice. In our tests, we never gave up the ambition of obtaining a proof of optimality. By necessity, there are surely limits to the scalability of this approach, but these should be regarded more as limits to the ambition of solving the general network design model to optimality rather than limits in the tested solution. The model and solution we have designed has successfully modeled and solved to optimality the situations form the applications we had in mind. Its efficiency allows to solve instances whose share size was beyond our original commitment.

To get a rough idea on the size of the instances that can be addressed, consider first the asymptotic growth of the number of variables that get allocated by our MILP formulation. Table 4.1 reports on the growth for each category of variables introduced.

Table 4.1: Number of allocated variables.

x	$O(\lvert\mathcal{N}\rvert\lvert\mathcal{T}\rvert\lvert\mathcal{Z}\rvert)$
y	$O(\lvert\mathcal{A}\rvert\lvert\mathcal{D}\rvert)$
γ	$O(\lvert\mathcal{D}\rvert\lvert\mathcal{T}\rvert)$
ρ	$O(\lvert\mathcal{T}\rvert^2)$
w	$O(\lvert\mathcal{N}\rvert\lvert\mathcal{T}\rvert^2)$
h	$O(\lvert\mathcal{A}\rvert\lvert\mathcal{D}\rvert^2)$
q	$O(\lvert\mathcal{A}\rvert\lvert\mathcal{Z}\rvert^2)$
j	$O(\lvert\mathcal{A}\rvert\lvert\mathcal{D}\rvert)$

Table 4.2: Number of defined constraints.

C.1	$O(\|\mathcal{N}\|\|\mathcal{Z}\|)$
C.2	$O(\|\mathcal{N}\|\|\mathcal{Z}\|\|\mathcal{T}\|)$
C.3	$O(\|\mathcal{A}\|)$
C.4	$O(\|\mathcal{A}\|\|\mathcal{D}\|)$
C.5	$O(\|\mathcal{N}\|\|\mathcal{T}\|^2)$
C.6	$O(\|\mathcal{A}\|\|\mathcal{D}\|^2)$
C.7	$O(\|\mathcal{N}\|\|\mathcal{Z}\|\|\mathcal{T}\|)$
C.8	$O(\|\mathcal{A}\|\|\mathcal{D}\|)$
C.9	$O(\|\mathcal{N}\|\|\mathcal{Z}\|\|\mathcal{T}\|)$
C.10	$O(\|\mathcal{A}\|\|\mathcal{D}\|)$
C.11	$O(\|\mathcal{T}\|)$
C.12	$O(\|\mathcal{A}\|\|\mathcal{D}\|^2)$
C.13	$O(\|\mathcal{A}\|\|\mathcal{D}\|^2)$
C.14	$O(\|\mathcal{N}\|^2\|\mathcal{T}\|^3)$
C.15	$O(\|\mathcal{T}\|^2)$
C.16	$O(\|\mathcal{A}\|\|\mathcal{D}\|^3)$
C.17	$O(\|\mathcal{D}\|\|\mathcal{T}\|)$
C.18	$O(\|\mathcal{A}\|\|\mathcal{D}\|^3)$
C.19	$O(\|\mathcal{A}\|\|\mathcal{D}\|^2)$

Those reported in the table are only worst case upper bounds as quite fewer variables get allocated in many instances; it is, however, easy to propose natural instance families meeting these bounds. Since allocating a variable takes $O(1)$ time and space, then the phase where the variables get introduced in the model one by one, through calls to the competent functions of the Gurobi dynamic library interface, takes

$$O(\max\{|\mathcal{N}||\mathcal{T}||\mathcal{Z}|, |\mathcal{D}||\mathcal{T}|, |\mathcal{N}||\mathcal{T}|^2, |\mathcal{A}||\mathcal{D}|^2, |\mathcal{A}||\mathcal{Z}|^2\})$$

time and space. Since inserting a constraint takes $O(1)$ time and memory for each one of its non-zero coefficients, the upper bound on the constraints specification phase can be similarly drawn from Table 4.2.

The max of these bounds works as an upper bound only for the model set up phase, whereas the true optimization phase managed by Gurobi requires further memory and may easily take exponential time. However, based on experiments and talking with reference to a desktop architecture, we are confident that these bounds may offer a rather good prediction on the ultimate performance of our code when considering to apply our implementation of the model, as it is, to other settings. More precisely, we are confident that these bounds may offer you a rather good prediction on the ultimate performance of our code over the limited range where the predicted memory consumption for the only allocation phase is not prohibitive.

4.5 Experimental Results

In this section, two case studies are presented with the aim of showing the expressiveness of the proposed design flow and the computational demand of the optimiza-

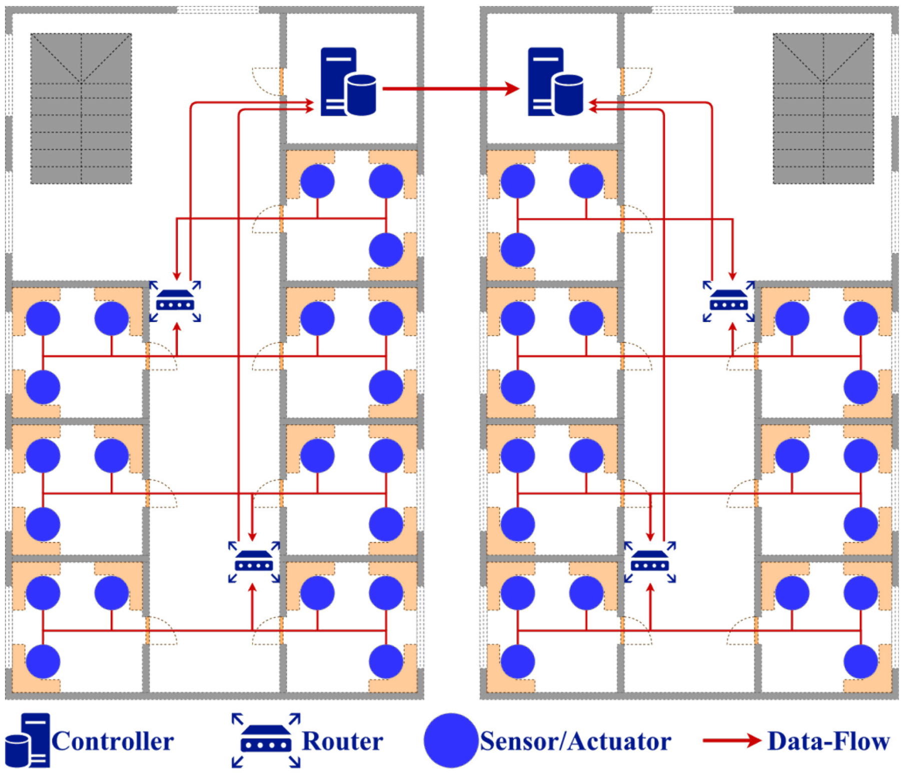


Fig. 4.4: Network topology with various network architectures connected through routers.

tion process. Network synthesis has been performed by using Gurobi 7.5.1 tool with Python 2.7.12 front-end on a 64-bit machine running Ubuntu 16.04 LTS; the machine features an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz with 16 GB memory.

4.5.1 Case Study 1

The first case study concerns the implementation of a distributed building automation application spanning over two adjacent buildings. The scenario is depicted in Figure 4.4. It consists of different kinds of tasks (*i.e.*, controllers, routers, sensors, and actuators), deployed inside rooms delimited by thick walls and exchanging a series of data-flows shown as red lines. Each building hosts a total of twenty-four tasks: a central controller, two routers and twenty-one sensors/actuators for room monitoring and regulation. Tasks are distributed over ten zones which comprise a control room, two technical closets for routers and seven offices. Each office contains two sensors and one actuator which have access to the central controller through an adjacent router. For this scenario, the technological libraries of available nodes and channels are reported in Table 4.3 and Table 4.4, respectively. The tables use the attribute symbols defined in Section 4.2.1.

Table 4.3: Catalog of available nodes for Case Study 1.

Label	*s*	*k*	*e*	*te*	*ek*	*m*
Development Board Type 1	32	5	5	1	0.10	*false*
Development Board Type 2	84	18	7	2	0.15	*false*
Development Board Type 3	64	22	8	2	0.30	*true*
Development Board Type 4	128	98	12	5	0.41	*true*
Development Board Type 5	256	128	15	6	0.33	*true*
Development Board Type 6	512	512	30	12	1.20	*false*

Table 4.4: Catalog of available abstract channels.

Label	*s*	*k*	*e*	*de*	*ek*	*dl*	*er*	*w*	*pp*
Bluetooth 4.0	24	9	1	1	0.16	12	10	*true*	*true*
IEEE 802.11ac	7000	34	3	2	0.30	8	7	*true*	*false*
IEEE 802.11ad	7400	79	7	4	0.28	3	4	*true*	*false*
Ethernet	200000	320	18	2	0.21	3	1	*false*	*true*
Fiber	273000	367	14	1	0.12	1	3	*false*	*true*

As explained in Section 4.2.1.d, an Abstract Channel encompasses the physical layer and all the required upper layers according to application scenario: in our case, the abstract channels in Table 4.4 include also TCP/IP. The presence of various network architectures allows to select the most appropriate one for each interconnection between nodes. Considering the environment depicted in Figure 4.4, the data-flow between the controllers has to cross a pair of thick walls which may hinder the use of wireless LAN. Such effect is represented as a lower value of conductance between the corresponding zones (*i.e.*, control rooms) for the wireless channels as reported in boldface in Table 4.5. Therefore, the optimization process will select a wired architecture even if it leads to higher cost for cabling which is computed by summing up dc attribute values for the given zone pairs and the chosen wired channel type (reported in boldface in Table 4.5).

In this example, the values in Tables 4.3, 4.4, and 4.5 have been obtained by performing a relative comparison between different technologies with the unique purpose of testing the optimization engine. More accurate values can be found in hardware datasheets for nodes and actual benchmarks for network architectures [94]. Contiguity relationship should be evaluated by the designer for each specific scenario, *e.g.*, by using well-known tools for WiFi deployment [95].

Vice versa, for other zones of the same scenario wireless communications, will be preferred for their lower cost. For instance, with energy minimization, we use 2 Bluetooth links, 7 IEEE 802.11ac links, 12 IEEE 802.11ad links, 10 Ethernet links,

Table 4.5: Example of contiguity values for Case Study 1.

z_1	z_2	ac	c	dc
Office 1	Technical closet 1	Bluetooth 4.0	0.45	0
		IEEE 802.11ac	0.58	0
		IEEE 802.11ad	0.97	0
		Ethernet	0.00	0
		Fiber	0.00	0
Technical closet 1	Control room 1	Bluetooth 4.0	0.12	0
		IEEE 802.11ac	0.46	0
		IEEE 802.11ad	0.84	0
		Ethernet	0.69	645
		Fiber	0.95	1,252
Control room 1	Control room 2	Bluetooth 4.0	**0.07**	0
		IEEE 802.11ac	**0.12**	0
		IEEE 802.11ad	**0.21**	0
		Ethernet	0.78	**824**
		Fiber	1.00	**1,486**

Table 4.6: Case Study 1: performance and results of the network synthesis as a function of the optimization objective.

Minimization Objective	CPU Time (s)	Economic Cost ($)	Energy Consumption (J)	Delay (s)	Error Rate (%)
Economic Cost	11.38	**38,903**	56,306	543.96	47.0
Energy	5.07	41,762	**48,330**	514.00	45.1
Delay	3.89	68,312	86,604	**283.35**	19.8
Error	3.97	67,760	93,809	287.84	**19.6**

and 5 fiber links. To the best of our knowledge, this is the first network synthesis approach that allows mixing different network architectures.

Table 4.6 shows the statistics of the synthesized networks with the four different optimization objectives presented in Section 4.3.3. The table reports the CPU time spent to find the optimal solution for a target optimization objective.

The composition of the synthesized network depends on the optimization metric. For instance, for cost minimization, the optimizer chooses 12 nodes of Type 2, 17 nodes of Type 3, and 9 nodes of Type 4, whereas, for energy minimization, it chooses 19 nodes of Type 2, 17 nodes of Type 3, and 9 nodes of Type 4. Furthermore, application tasks can be grouped in different ways according to task-node assignment

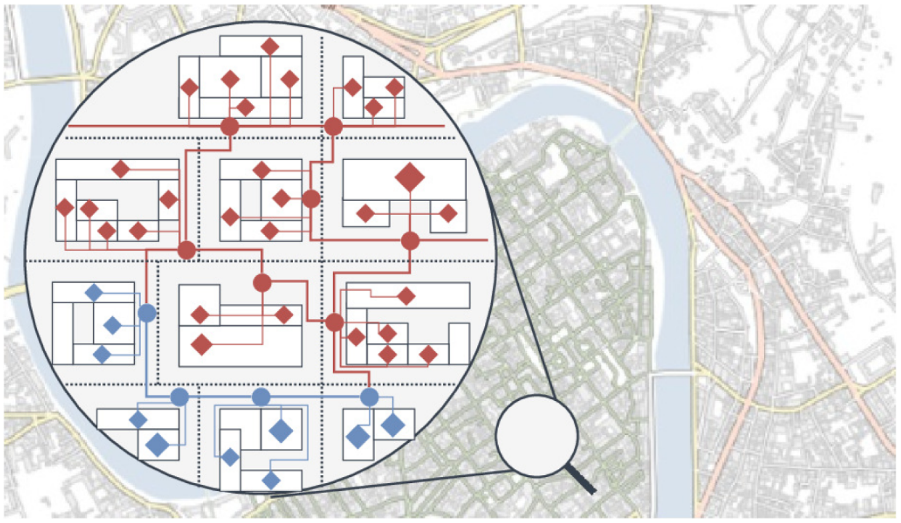


Fig. 4.5: Wide urban area test case: the tasks and data-flows colored in blue represent a pre-existing network while red elements have been added during network synthesis.

which is determined by the optimization process. For instance, let us consider three sensor/actuator tasks, denoted as $T1$, $T2$, and $T3$, belonging to the same zone (*i.e.*, room). With economic cost minimization, all the tasks are placed inside the same Type 4 node whereas, with delay minimization, two of them are placed inside inside distinct nodes of Type 3 while the third is placed inside a node of Type 4.

In summary, this Case Study shows that:

- different network architectures can be mixed in the synthesis process;
- a single heavy application can be distributed over multiple nodes of a distributed CPS;
- the optimization process can distribute tasks in different ways over the network according to the optimization objective.

4.5.2 Case Study 2

The second case study concerns the implementation of a smart city application. For example, energy efficiency is a well-known design problem in this context [96]. The description of the environment for the proposed case study is given by the cartography in Figure 4.5. The area is subdivided into zones delineated by dotted lines. Each zone contains a task named *distributor* (represented by a circle) and a variable number of *user* tasks, all connected through data-flows. In such a vast public context, it is quite common to exploit a pre-existing network and add new pieces of infrastructure. This fact gives us the opportunity to show how the proposed synthesis flow can handle this kind of constraint. By referring to Figure 4.5, the tasks and data-flows colored in blue represent a pre-existing network, *i.e.*, they have already been placed inside nodes and channels, respectively. Vice versa, the red tasks, and data-flows are assigned to nodes and channels by the network synthesis process. The catalogs of nodes and channels are shown in Table 4.7 and Table 4.4, respectively. The contiguity values between zones are mainly dependent on their distance.

Table 4.7: Catalog of available nodes for Case Study 2.

Label	*s*	*k*	*e*	*te*	*ek*	*m*
Development Board Type 1	64	10	2	1	0.05	*false*
Development Board Type 2	98	24	4	2	0.15	*true*
Development Board Type 3	128	64	8	4	0.40	*true*
Development Board Type 4	256	128	14	7	0.32	*true*
Development Board Type 5	512	378	20	10	0.60	*false*

This case study aims at evaluating the scalability of the approach as a function of scenario size, *i.e.*, number of zones, tasks, and data-flows. For this purpose, we automatically generated instances with increased size by using two different approaches.

4.5.2.a Scalability over zones

The first scalability test regards the creation of large scenarios by increasing the number of zones, while the number of tasks per zone remains quite small. The generation of instances is based on the following rules:

- zones are arranged as a chain, and a new instance is automatically generated by increasing the chain;
- each zone has a set of contiguity values defined only for the precedent zone and subsequent zone unless it is the first or the last zone of the chain;
- each zone contains four user tasks and one distributor;
- in each zone four data-flows are connecting each user task with the distributor of the zone;
- the distributor of a zone is connected to the distributor of the precedent zone and the one of the subsequent zone unless it belongs to the first or last zone of the chain;
- attributes of tasks and data-flows are constant, and their value depends on the role of the task, *i.e.*, distributor or user, and on the type of connection, *i.e.*, between distributors or between user and distributor.

Figure 4.6 shows the value of the objective function as a function of the size of the input instance (number of tasks). Even if values have been normalized to fit in the same plot, each of them is the minimum when the corresponding objective function is used to drive the optimization. Therefore, we can conclude that the behavior of the synthesizer is consistent over a large set of problem instances.

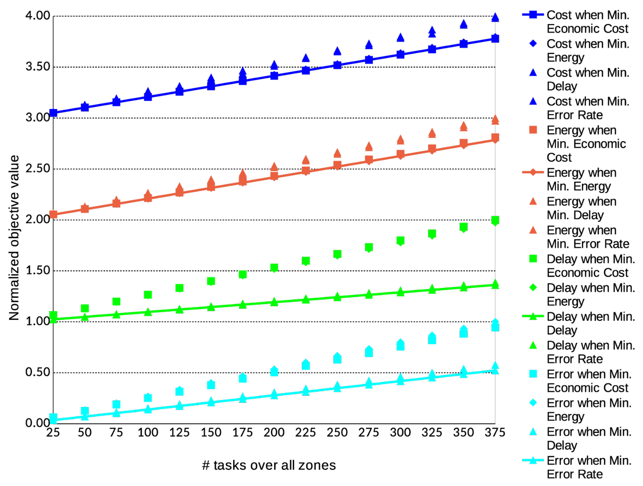


Fig. 4.6: Value of the objective function as a function of number of tasks over all zones and the optimization objective.

Figure 4.7 shows the total optimization time for the synthesizer as a function of the size of the input instance (number of tasks over all zones) for all optimization targets. Time values have been computed by using Python `time.clock()` function

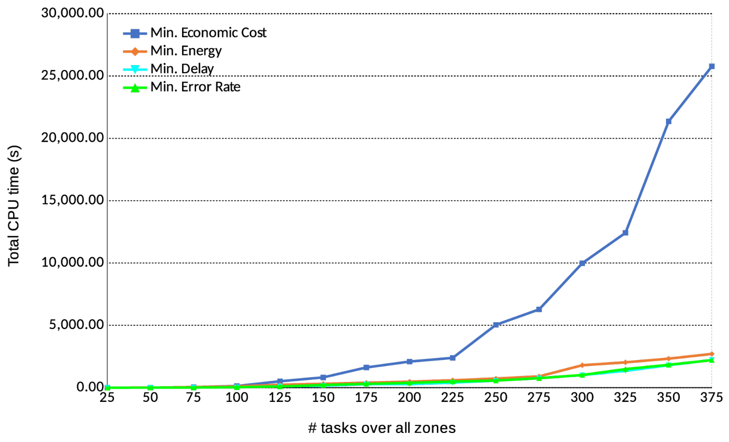


Fig. 4.7: Total CPU time for Case Study 2 as a function of number of tasks over all zones.

which takes into account the effort spent by the CPU in each thread of the process[1] thus avoiding artifacts due to the current load of the workstation and hyper-threading techniques. The graph trend of the objective functions in Figure 4.7 can be directly related to the algorithmic complexity and the number of involved variables of the

[1] https://www.pythoncentral.io/measure-time-in-python-time-time-vs-time-clock

objective functions reported in Section 4.3.3. Even if the economic cost and energy consumption minimization functions have the same number of loops, the former has a higher number of involved variables. As such, the economic cost minimization requires more effort than the other functions as shown in the graph.

As described in Section 4.4, the synthesis process consists of some steps which prepare the proper optimization phase, *i.e.*, parsing of the instance description file, generation of variables, and generation of constraints. It is worth analyzing how CPU time is spent in this optimization flow. Figure 4.8 shows the percentage of the total optimization time (reported in Figure 4.7) devoted to pre-optimization activities. For economic cost minimization, the real optimization phase is mainly predominant over preparation but for simpler objective functions and small instances, preparation time can be higher than time spent to search the optimal solution.

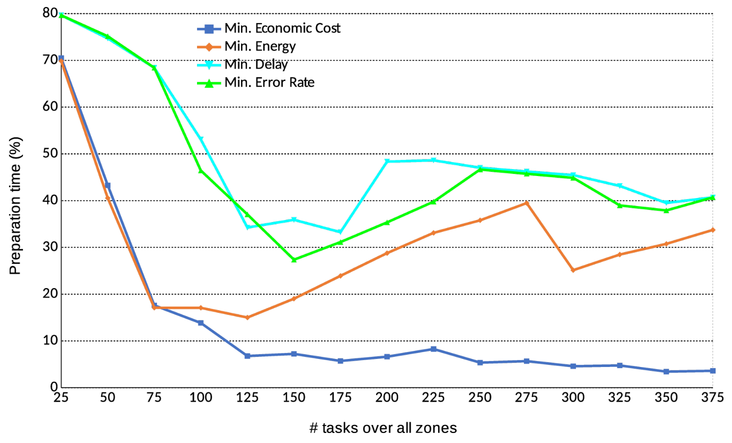


Fig. 4.8: Percentage of CPU time devoted to pre-optimization activities for Case Study 2 as a function of number of tasks over all zones.

To complete the scalability analysis, Figure 4.9 shows the corresponding memory usage.

4.5.2.b Scalability over tasks

The second scalability test regards the creation of large scenarios by increasing the number of tasks in the same zone. The generation of instances is based on the following rules:

- only one large zone is considered;
- no contiguity relationships are set;
- each zone contains an increasing number user tasks and one distributor;
- in each zone there are as many data-flows as user tasks since they connect each user task with the distributor of the zone;

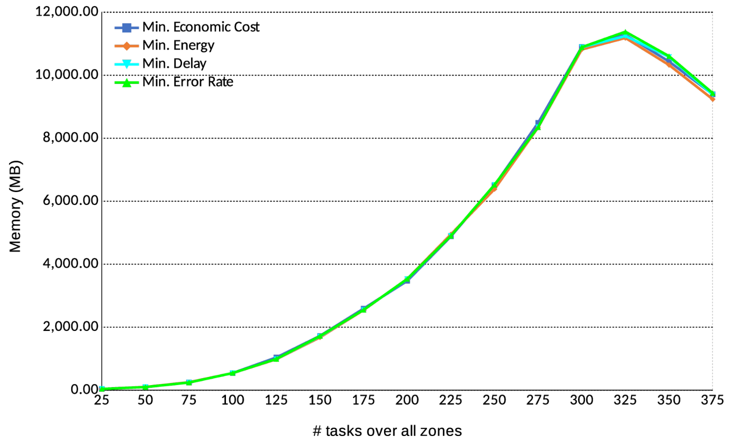


Fig. 4.9: Memory usage for Case Study 2 as a function of number of tasks over all zones.

- attributes of tasks and data-flows are constant, and their value depends on the role of the task, *i.e.*, distributor or user, and on the type of connection, *i.e.*, between distributors or between user and distributor.

Figure 4.10 shows the total optimization time for the synthesizer as a function of the number of tasks per zone. We only show economic cost minimization since the previous analysis proved it to be the most computationally intensive target. Figure 4.11 shows the corresponding memory usage.

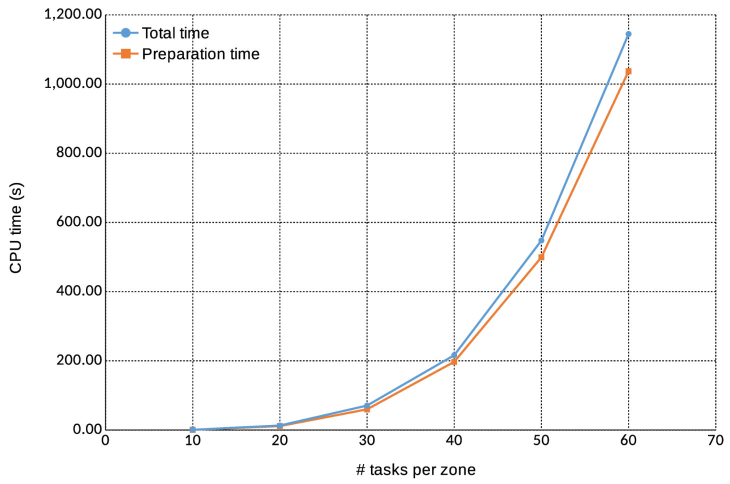


Fig. 4.10: CPU time for Case Study 2 as a function of the number of tasks per zone.

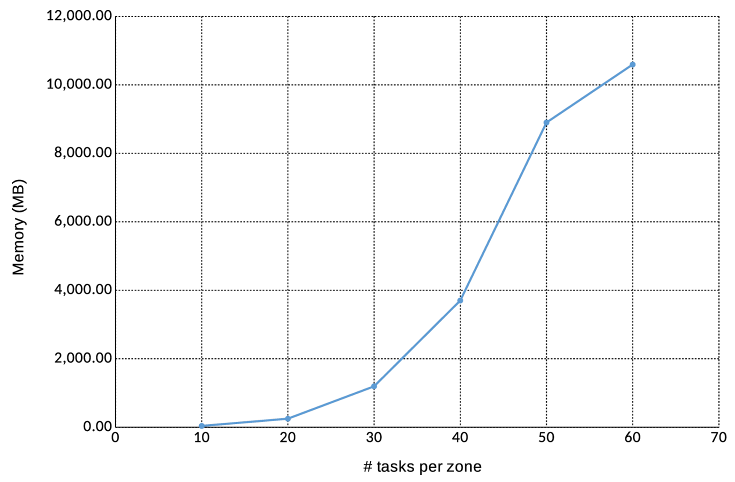


Fig. 4.11: Memory usage for Case Study 2 as a function of the number of tasks per zone.

4.6 Conclusions

This work focused on the aspects of Networked Cyber-Physical Systems (NCPSs) and proposed an extended design flow to address them. Assuming that highly optimized nodes are desirable, the network infrastructure should be decided before designing hardware and implementing software thus leading to the concept of network synthesis. A communication-aware formalization was proposed to specify constraints and optimization metrics. Network synthesis was formalized as an optimization problem using mixed-integer linear programming. We defined the following formal entities: tasks, data-flows, nodes, abstract channels, and zones. The last two entities are particularly innovative. The Abstract Channel generalizes the concept of network architecture (*i.e.*, physical channels and protocols) so that the final solution can combine different types of network architectures. The Zone generalizes the concept of physical location adapting location accuracy to the requirements of the application and focusing on the impact of node placement on communications and cost. The framework was applied to real case studies with the aim to show the advancement with respect to state of the art. The first case study shows the possibility to create network infrastructures containing different network architectures according to users' needs and environmental constraints. The second one highlights the possibility to synthesize a network by adding components to an existing infrastructure which is a common problem in real life scenarios. Future work aims at investigating the scalability issues of the Mixed Integer Linear Programmings (MILPs) approach and proposing communication-aware heuristics to address huge problems.

5

Fault modeling and injection

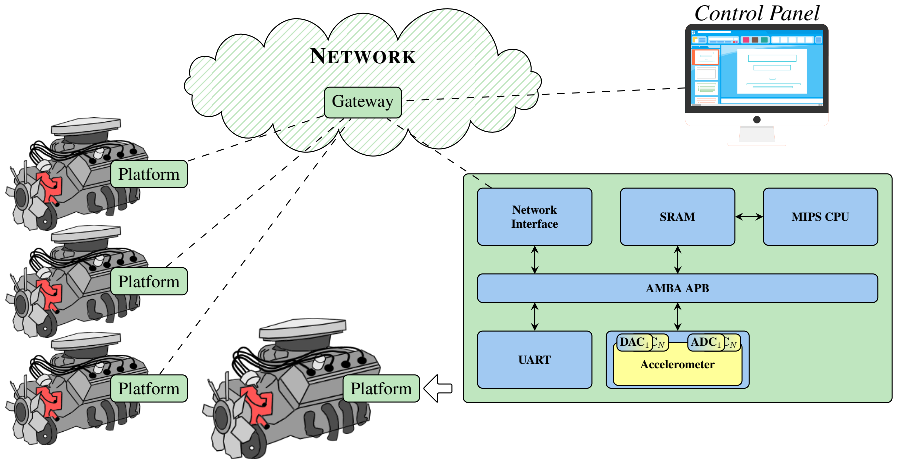


Fig. 5.1: Architecture of a case study for predictive maintenance.

The previous chapters show how we propose to build a holistic Networked Cyber-Physical Systems (NCPSs), from the abstraction of analog components, to the synthesis of network architectures. Networked Cyber-Physical Systems (NCPSs) are indeed at the base of revolutionary phenomena as Industry 4.0 which represents an integration of Internet-of-Things (IoT) and relevant physical technologies, including analytics, additive manufacturing, robotics, high-performance computing, artificial intelligence and cognitive technologies, advanced materials, and augmented reality.

Figure 5.1 shows a case study concerning predictive maintenance based on vibration monitoring through wireless accelerometers [97]. Vibration is a natural phenomenon in every industrial facility that causes wear and tear on machine parts, leading to equipment failure. Active vibration monitoring and control by analyzing the vibration patterns of machinery can prevent such occurrences, which in turn improves overall system performance, efficiency, lifetime, and safety. The figure reports a set of machines each equipped with wireless accelerometers (denoted as "network nodes" in the following) transmitting vibration data through a gateway to the cloud for the estimation of the probability of short-term failure by using statistical and artificial intelligence techniques. Each wireless sensor is a networked cyber-physical system consisting of *digital* components, *e.g.*, CPU and memory, *analog* components, *e.g.*, the accelerometer, and a *network* interface.

Ensuring functional safety of such systems is becoming a complex and critical task [98] which needs an effective and reliable procedure which can assess their correctness. One solution is to move the safety verification to a higher abstraction level and earlier in the development process [99, 100]. This would enable the use of well-established procedures to coordinate the processes of fault injection. For *digital* systems, solutions can be found at different levels of abstraction, *e.g.*, gate-level [101], Register-Transfer Level (RTL) [102] and also at Transaction-Level Modeling (TLM) [98]. While such systems are very heterogeneous being composed of digital hardware, analog hardware and networks, the current functional safety assessment is mainly focused on digital hardware. Minor attention is devoted to analog hardware [103] and not at all to the interconnecting network.

In networked cyber-physical systems, the dependability must be verified not only for the nodes in isolation but also by taking into account their interaction through the communication channel. Considering the example of Figure 5.1, a set of network nodes interacting through the network to perform a distributed task is not so different from a set of cores in a system-on-chip or a network-on-chip. This fact suggests considering *the whole distributed application as a single system to be verified* rather than each component node in isolation. By following this approach, inter-node interactions can be considered as those among cores in a system-on-chip. Node interactions can be very complex and dependent on the application; they range from simple point-to-point communications, *e.g.*, to collect data from sensor nodes, to many-to-many communications as in case of fully distributed algorithms, *e.g.*, consensus protocols. This discussion leads to the need for performing safety assessment not only on digital hardware but also on analog hardware and communications subject to electromagnetic interferences, packet collisions, and traffic congestion.

Designers often rely on different languages which may not be conceived to work together when mixing digital, analog, and network models. This incompatibility makes functional safety evaluation more difficult in networked cyber-physical systems and the development of reliable safety mechanisms even more difficult.

Figure 5.2 shows how this work proposes to solve these problems by combining different domains into a single *holistic* platform. The keywords here are: abstraction and injection. The former means the simplification of the initial description for better and easier integration (*i.e.*, C++). The latter means the instrumentation or manipulation of the code for fault injection. This could take place at different stages of the manipulation process. For instance, for the digital and network domains the injection takes place directly in the abstracted description, while for the analog domain it takes place at circuit-level before the abstraction. Ultimately, this thesis also aims at pre-

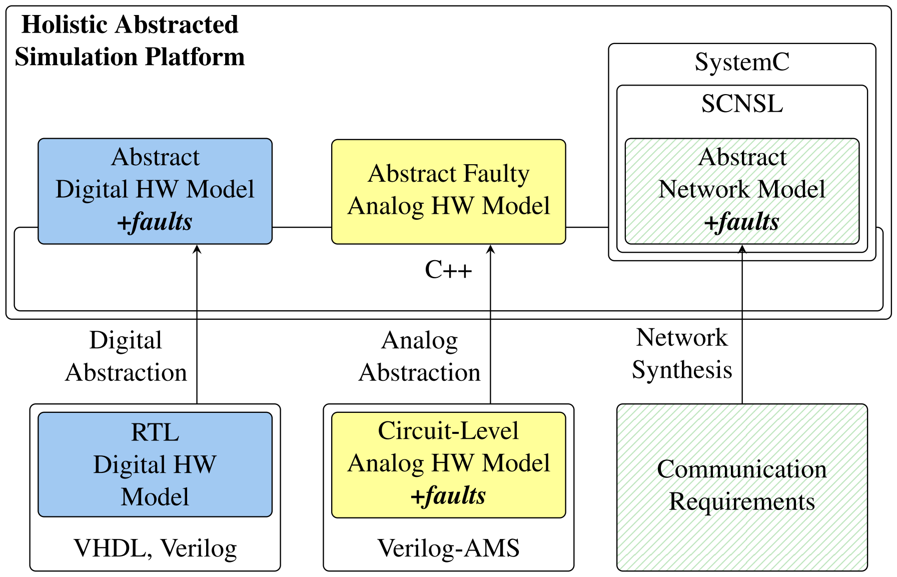

Holistic Abstracted Simulation Platform
SystemC
SCNSL
Abstract Digital HW Model +faults
Abstract Faulty Analog HW Model
Abstract Network Model +faults
C++
Digital Abstraction
Analog Abstraction
Network Synthesis
RTL Digital HW Model
VHDL, Verilog
Circuit-Level Analog HW Model +faults
Verilog-AMS
Communication Requirements

Fig. 5.2: Generation of the holistic platform for fault simulation.

senting an effective way of injecting faults into the different domains of a networked cyber-physical system for functional safety evaluation.

This chapter, is organized as follows: Section 5.1 describes the methodology for high-level fault injection in the digital domain. Section 5.2 first presents a taxonomy of the analog fault models considered by this thesis, and then describes how the injection is performed in the analog domain. Finally, Section 5.3 describes the fault models for the network communications, and how they are activated during the simulation. The experimental results are thoroughly presented in the next chapter.

5.1 Digital fault injection

The digital fault injection methodology presented in this section instruments the code with mutants that simulate fault models at the bit level (*e.g.*, stuck-at, bit-flip). As shown in Figure 5.2 the starting description for the digital fault injection process is a design modeled at RTL. We have seen in Chapter 3, Section 3.1.3, how these descriptions are then abstracted with the methodology presented in [38] to allow the integration with the analog components. After applying the abstraction, the digital fault injection produces an output description containing all the mutations that can reproduce the faulty design behavior.

A previous work presented in [104] proposed a methodology injecting mutants by using type-dependent mutation functions and controlling the injection through HDL ports. There are several drawbacks with that approach, which are: 1) the sensible overhead introduced by relying on complex injection functions and HDL ports,

2) the lack of generality of such functions which must be implemented for each datatype, and 3) it does not take into account hierarchical designs. This work overcomes these limitations by proposing an alternative which relies on *bitwise* operations and compile-time optimized injection functions.

With the approach proposed in [104], most of the time is spent by calling the injection functions and the consequent operations: saving the stack, updating the stack pointer, allocating the memory for the variables inside the function and copying its parameters. The return phase of a function requires additional operations, such as: restoring the stack and copying the returned value. Furthermore, the injection is performed by writing on the *fault_port*, and during the simulation by reading its value in order to determine if the fault is currently active. These writes and especially read operations further decrease simulation efficiency. Another critical problem of that approach concerns the injection inside hierarchical designs, especially when a model is instantiated multiple times. Each location is identified by the fault-free value, and the injection range identified by the pair of variables *start_range* and *end_range*. The range controls the activation of a fault whenever the currently active fault index is inside it. However, if a model is instantiated multiple times, the same ranges appears on each one of its instances. As a consequence, activating a single fault could actually enable multiple faults.

The approach proposed in this thesis controls the activation/deactivation of a fault through the pair of variables *instance_number* and *fault_number*. The former controls which instance of the design needs to be injected, while the latter instead controls which fault is currently active inside that instance. During the instrumentation of the code, each instantiated design is numbered with a unique id called *_id*. During the simulation, this id is compared against *instance_number* to determine if an active fault is inside the current instance. This is a simple solution which allows injecting also into hierarchical designs. The proposed approach exploits as much as possible the capabilities and optimizations introduced by modern compilers to reduce the instrumentation overhead, especially by avoiding the use of complex injection functions. The idea is to perform *bit level* fault injection by using *bitwise* operations and functions which are replaced/resolved at compile-time.

As a *first solution* we propose to instrument the code with preprocessor commands called *macros* provided by the C++ language. The compiler, before the actual compilation of code, performs the so-called macros “expansion”, replacing a macro call with the processed copy of its body. This solution allows instrumenting the code and avoids function calls. Let us now examine into details the first two macros of Listing 5.1, *is_active* and *inject*. The former macro accepts three arguments, the instance number of the component where the fault resides, the fault location, and the dimension of the location (*e.g.*, the number of bits). Based on this three arguments, the macro builds the expression which returns *true* whenever the current active fault resides in the given instance at the given location. The latter macro contains a ternary operator which, calls the injection *function* if the current fault is active, or returns the fault-free value if inactive. The three macros, *rise_bit*, *clear_bit*, and *toggle_bit* are used as support to implement bit level fault injections.

Listing 5.1: Mutation macros implemented through bitwise operations.

```
// Injection control variables.
uint32_t instance_number = 0UL;
uint32_t fault_number    = 6UL;
// Checks if the fault is active.
#define IS_ACTIVE(instance, location, dimension) \
    ((instance == instance_number) && \
    (fault_number >= location) && (fault_number < (location + dimension)))
// Performs the injection.
#define INJECT(instance, location, dimension, value, function) \
    (IS_ACTIVE(instance, location, dimension) \
        ? function(value, fault_number - location) : value)

// Rises the bit at the given position.
#define RISE_BIT(position) \
    (1 << (position))
// Clears the bit at the given position.
#define CLEAR_BIT(value, position) \
    (value & ~RISE_BIT(position))
// Toggles the bit at the given position.
#define TOGGLE_BIT(value, position) \
    (value ^ RISE_BIT(position))

// Defines which type of stuck-at should be injected (0 or 1).
uint32_t sa_type = 1UL;
// Injects a stuck-at fault at the given position.
#define STUCK_AT(value, position) \
    ((-sa_type & RISE_BIT(position)) | CLEAR_BIT(value, position))
```

For instance, the *stuck_at* macro is used to inject **stuck-at** fault and relies on the two *rise_bit* and *clear_bit* macros, as well as the variable *sa_type* which determines which type of stuck-at fault must be injected (*i.e.*, 0 or 1). Let us see how the expression contained in the *stuck_at* macro is able to inject a stuck-at-1 fault on the 4-th bit of an 8-bit variable which has an initial value of 37:

```
uint8_t value  = 37;                 // 0010 0101
uint8_t exp1   = CLEAR_BIT(value, 4); // 0010 0101

uint8_t exp2a  = -sa_type;           // 1111 1111
uint8_t exp2b  = RISE_BIT(4);        // 0001 0000
uint8_t exp2   = (exp2a & exp2b);    // 0001 0000

uint8_t result = (exp1 | exp2);      // 0011 0101
```

Another example of bit level fault is the **bit-flip**, which can be injected by using the *toggle_bit* macro shown in Listing 5.1.

This first solution for bit level fault injection sensibly reduces the instrumentation overhead by relying on bit-wise operators, and provides a type-independent injection infrastructure. However, it relies on macros lacking a type-checking system, which can be a valuable support for ensuring the correctness of the simulation. The *second solution* is similar to the first one in that they both exploit bit-wise operators, but instead of using macros it relies on *inline template* functions. This new set of injection functions is shown in Listing 5.2. Marking a function *inline* tells the compiler that every call to it can be replaced with its body instead of going through the process of “function call”. However, marking a function *inline* is a *suggestion* to the compiler

Listing 5.2: Mutation functions implemented through bitwise operations.

```
// Injection control variables.
uint32_t instance_number = 0UL;
uint32_t fault_number    = 6UL;
// Checks if the fault is active.
inline bool is_active(uint32_t ins, uint32_t loc, uint32_t dim) {
    return (ins == instance_number) &&
           (fault_number >= loc) && (fault_number < (loc + dim));
}
// Performs the injection.
template<typename T, typename Tf>
inline T inject(uint32_t ins, uint32_t loc, uint32_t dim, T val, Tf fun) {
    return is_active(ins, loc, dim) ? fun(val, fault_number - loc) : val;
}

// Rises the bit at the given position.
template<typename T>
inline T rise_bit(T pos) {
    return 1 << pos;
}
// Clears the bit at the given position.
template<typename TVal, typename T = uint32_t>
inline TVal clear_bit(TVal value, T position) {
    return value & ~rise_bit<T>(position);
}
// Toggles the bit at the given position.
template<typename TVal, typename T = uint32_t>
inline TVal toggle_bit(TVal value, T position) {
    return value ^ rise_bit<T>(position);
}

// Defines which type of stuck-at should be injected (0 or 1).
uint32_t sa_type = 1UL;
// Injects a stuck-at fault at the given position.
template<typename TVal, typename T = uint32_t>
inline TVal stuck_at(TVal val, T pos) {
    return ((-sa_type & rise_bit<T>(pos)) | clear_bit<T, TVal>(val, pos));
}
```

which can actually ignore the request based on the number of instructions inside the function. Thanks to the simple design of the bit-wise injection function, these are always replaced. This solution relies on template arguments which allow reducing the number of declared functions without losing the safety of a type-checking system [105]. Templates are a C++ feature that allows parameterizing the code (*e.g.*, parameters, functions, *etc.*) *at compile time*, thus resulting in faster execution of the code.

Let us now look at an example of high-level abstracted code where the two approaches are applied. Listing 5.3 shows the abstracted C++ code of the Analog-to-Digital Converter (ADC) shown in Listing A.1. The upper part shows the non-instrumented code, the middle one shows the code instrumented with macros, while the lower part shows the same code injected with the inline template functions. In this example the *toggle_bit* has been used to inject bit-flip faults on two if guards. The two instrumented codes are almost identical, except that the code injected with inline functions requires to specify the type of the function (*i.e.*, *bool*, *int32_t*, *etc.*).

Listing 5.3: Sample code of an analog to digital converter. On the top the original code, in the middle the same code injected with C++ macros, while on the bottom the injection with inline template functions.

```
void process_adc() {
    double sample = vin;
    bool over = false;
    int32_t i = (bits - 1);
    while (i >= 0) {
        over = (sample >= midpoint);
        if (over) {
            sample -= midpoint;
        }
        sample *= 2;
        out[i] = over;
        i = i - 1;
    }
}
```

```
void process_adc() {
    double sample = vin;
    bool over = INJECT(_id, 0, 1, false, STUCK_AT);
    int32_t i = INJECT(_id, 1, 8, bits - 1, STUCK_AT);
    while (INJECT(_id, 9, 1, i >= 0, TOGGLE_BIT)) {
        over = INJECT(_id, 10, 1, sample >= midpoint, STUCK_AT);
        if (INJECT(_id, 11, 1, over, TOGGLE_BIT)) {
            sample -= midpoint;
        }
        sample *= 2;
        out[i] = INJECT(_id, 12, 1, over, STUCK_AT);
        i -= INJECT(_id, 13, 8, 1, STUCK_AT);
    }
}
```

```
void process_adc() {
    double sample = vin;
    bool over = inject(_id, 0, 1, false, stuck_at<bool>);
    int32_t i = inject(_id, 1, 8, bits - 1, stuck_at<int32_t>);
    while (inject(_id, 9, 1, i >= 0, toggle_bit<bool>)) {
        over = inject(_id, 10, 1, sample >= midpoint, stuck_at<bool>);
        if (inject(_id, 11, 1, over, toggle_bit<bool>)) {
            sample -= midpoint;
        }
        sample *= 2;
        out[i] = inject(_id, 12, 1, over, stuck_at<bool>);
        i -= inject(_id, 13, 8, 1, stuck_at<int32_t>);
    }
}
```

5.2 Analog fault injection

The idea presented in this section combines the abstraction process with an automated fault injection flow. Figure 5.3 exemplifies the structure of this fault analysis flow. Based on the types of fault that we are interested in, the injection process uses the circuit topology to to generate a list of viable locations for *saboteurs* automatically. Then, a new faulty description is generated for each fault location by injecting the equations describing the desired fault model. This is applied to all the

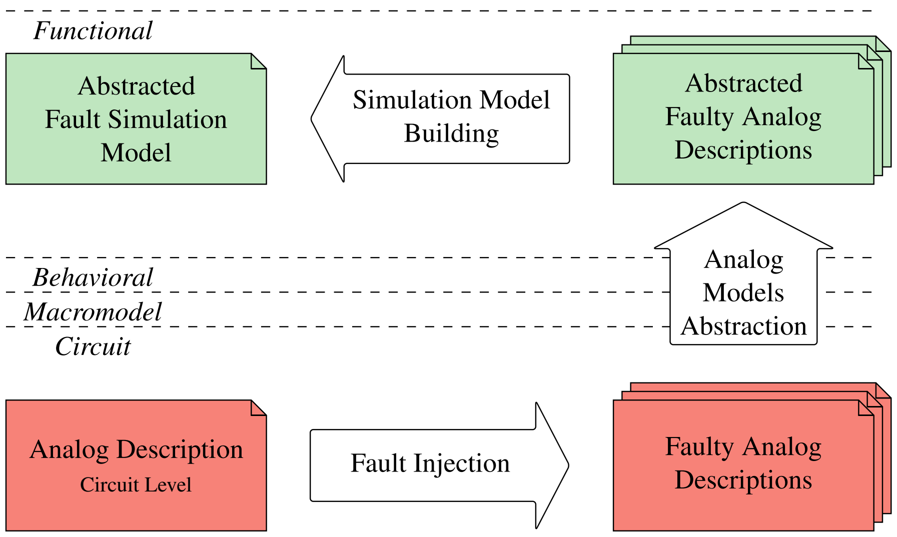

Functional
Abstracted Fault Simulation Model
Simulation Model Building
Abstracted Faulty Analog Descriptions
Analog Models Abstraction
Behavioral
Macromodel
Circuit
Analog Description
Circuit Level
Fault Injection
Faulty Analog Descriptions

Fig. 5.3: Analog fault injection flow.

fault locations. Each faulty description is then manipulated using the analog abstraction methodology presented in the previous chapters. Finally, all the faulty abstracted descriptions are recombined to build the simulation model.

This section starts with a taxonomy of the most notable analog fault models taken from the literature and widely used by commercial tools for fault injection. It provides the code manipulations required to injected the faults in behavioral-level descriptions written in Verilog-AMS. Then, it introduces the idea of fault locations mapping from the transistor to the behavioral-level. Moreover, it shows how the final C++ description is built for simulating all the analog faults.

5.2.1 Fault taxonomy and code manipulation

Follows a collection of analog fault models taken from the literature, and categorized into a multi-level taxonomy to show their relations with each level of abstraction. As such, each fault is exemplified with a piece of Verilog-AMS code. The aim of this section is twofold: 1) clarify which faults are considered by the proposed methodology, and 2) how they are injected at the behavioral level.

This methodology focuses on the following types of faults: *short-circuit/bridge*, *open-circuit*, *potential/flow pulses* and *parametric* faults. The first three fault models are called *saboteurs* and are usually injected by employing parametrized analog blocks. As shown in the following sections, they are injected at the behavioral level by adding a new equation to the model. The fourth type of fault, the parametric fault, are called *mutants* and produce small deviations or mutations of component parameters, and for this reason they are the hardest to detect.

Table 5.1 shows a taxonomy of the aforementioned fault models based on the four analog abstraction levels proposed in [1]. In detail, a *circuit* description is usually de-

Table 5.1: Proposed taxonomy of analog fault models at different levels of abstraction.

Abstraction Level	**Fault Model**			
	Bridge/Short Circuit	Open Circuit	Potential/Flow Pulse	Parametric
Functional			✓	✓
Behavioral			✓	✓
macro-model	✓	✓	✓	✓
Circuit	✓	✓	✓	✓

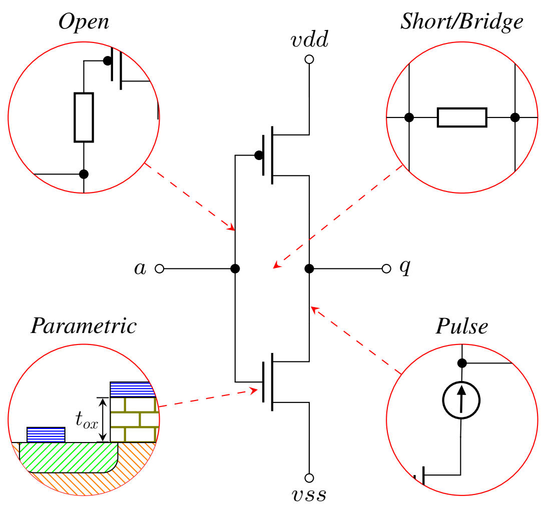


Fig. 5.4: Locations of possible analog faults of a transistor-level description of an inverter.

fined by connecting SPICE primitives and must abide by the laws of conservation of energy. An example of a circuit description is the inverter shown in Figure 5.4. A *macro-model* description is a simplified circuit made of controlled sources which cannot be associated with an actual circuit but which satisfy the laws of conservation of energy. A *behavioral* description is a mathematical description with no internal structure and which satisfies the laws of conservation of energy but only at its interfacing pins. A *functional* model is a mathematical signal flow description, which has no internal structure and which is not compelled to abide by the laws of conservation.

5.2.1.a Short/Bridge

The first type of fault we are going to analyze is the *short*, usually caused by two bare wires in a circuit which touch each other. Besides the different configurations that a *short* can assume, it can be considered a particular case of a *bridge* fault with a near-zero resistance value. *Bridge* faults usually assume values which range from 0Ω to $10\text{k}\Omega$ [106, 107].

For the injection at the circuit and macro-model levels, a *short* can be modeled as a relatively small resistor. An example of a short circuit is shown on the top-right corner of Figure 5.4. The depicted circuit can be injected with a total of six *bridge* faults, one between each node of the circuit. This is the estimated number of faults without taking into account the different resistance values that each *bridge* could assume. This type of fault is generally used at the circuit and macro-model levels since it requires knowledge about the topology of the design. However, with a good insight into the model, or by means of previous analysis it is possible to represent such a fault at the behavioral level [108].

The *short* fault can be modeled in Verilog-AMS with the following line:

```
I(a, q) <+ V(a, q) / 1;
```

5.2.1.b Open Circuit

Open fault models are saboteurs generated by missing contacts or cracks on the interconnections of a circuit. However, contrary to what one might think, *opens* do not entirely block the current flowing through an edge of the circuit, rather, they dramatically increase the resistance of it [109]. This fault model is generally used at the circuit and macro-model levels, for the same reasons as *short* circuits. This fault can be injected by adding a high resistance in series with an existing edge of the circuit.

An example of an *open* circuit is shown on the top-left corner of Figure 5.4. The *open* fault can be modeled in Verilog-AMS with the following line:

```
V(a, m1g) <+ I(a, m1g) * 1e06;
```

5.2.1.c Potential/Flow Pulses

Potential and flow pulses are components commonly used to inject the class of saboteurs modeling Single Event Transients (SETs). This kind of fault is physically generated by alpha particles or neutrons hitting a sensitive node of the circuit. Approaches to inject this kind of fault at the behavioral level in VHDL-AMS and Verilog-AMS can be found in [110] and [111], respectively.

An example of *pulse* fault is shown on the bottom-right corner of Figure 5.4. Such a fault can be modeled by a controlled potential/flow source and can be described in Verilog-AMS with the following equation:

```
I(m2d, q) <+ Pulse;
```

where *Pulse* is a value controlled by the simulation environment and updated at each simulation step to match the waveform of the desired fault model (*e.g.*, double exponential, damped sinewave).

5.2.1.d Parametric

Parametric faults can have many causes, *e.g.*, additional residues of metal during manufacturing of an Integrated Circuit (IC) or even deterioration due to aging. We can consider *parametric* faults to be all of those that make the design parameters fall outside acceptable boundaries. It is infeasible to test all the possible variations of the values of such parameters since they belong to the domain of real numbers and, thus, can assume an infinite number of values. As a consequence, many works which try to deal with this type of faults have developed techniques which aim at reducing the number of faults that have to be tested [112, 113].

Let us take the circuit of Figure 5.4 and in particular the NMOS. Parametric faults can be injected by modifying the components instantiation. For instance, we could modify the parameter controlling the oxide thickness tox:

```
nmos #(.tox(10u), ...) N1(nD, nG, nS, nB);
```

Even though the previously proposed approaches reduce the number of faults to test, the problem of simulating all the “selected” parameters deviations endures. The analog abstraction methodology is especially important for this reason, because it enables simulating all these combinations in less time.

5.2.2 Transistor-level to Behavioral-level fault mapping

Let us now consider the transistor-level description of the inverter circuit shown in Figure 5.4, and inject it with a set of open-circuit and short-circuit faults typically used for MOSFETs (*i.e.*, gate, drain, and source opens and shorts). The locations which are possibly subject to fault injection are shown in Figure 5.5, identified by a numerical id.

The idea is to map the information about the identified locations to the higher level, to obtain the same faulty behavior at different abstraction levels. To exemplify the concept, Figure 5.6 shows the same injection locations inside the behavioral-level description, where IDs matches the ones in Figure 5.5. For fault injection is vital that each location identified through this mapping are preserved even in the subsequent steps of the methodology, in particular during the abstraction presented in the next section.

Until now the example shows how the methodology can be applied to catastrophic opens and shorts. However, the vast majority of faults are represented by unexpected parameter deviations. As explained in the previous section, the problems rising are mainly two: 1) there are a considerable number of possible deviations, and 2) fault simulation is one of the main tools used to test them. Section 3.6 shows how the state-dependent behavioral models can be tuned to fit the physical level properties of the original transistor-level circuits. As such, mismatches can be captured by this high-level descriptions by manipulating its parameters, for instance, variable `C_a_vdd_h` appearing in the Verilog-AMS code of the inverter shown in Listing A.7. It is worth noting that, with the proposed abstraction methodology the effort required to inject parametric faults is lesser than the one required for opens,

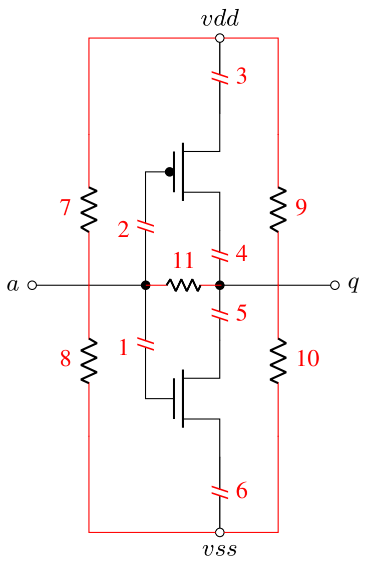


Fig. 5.5: Transistor-level inverter with locations of possible open-circuit and short-circuit faults.

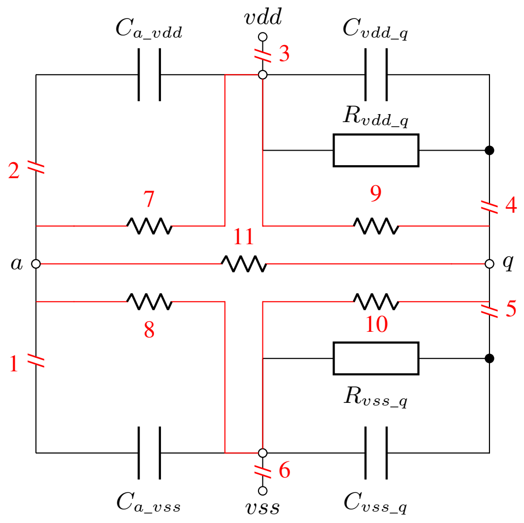


Fig. 5.6: Behavioral-level inverter with locations of possible open-circuit and short-circuit faults.

Listing 5.4: C++ interface of a simulation model.

```
class ISimulationComponent {
public:
    virtual void process(double ts) = 0;
    virtual void update_interfaces() = 0;
};
```

shorts, and pulses, because their injection inevitably changes the circuit structure. The abstraction is built upon the generalized version of Kirchhoff's laws, which is required to build the equations for loops and nodes. As a consequence, changing the circuit structure means re-computing these equations, and re-applying the abstraction flow. This is the injection flow, shown in Figure 5.3, produces a set of *faulty analog descriptions*.

5.2.3 Preserving faulty behaviors during abstraction

The process of abstraction to the functional-level must not remove the introduced equations or the manipulated parameters required to model the injected fault. The equations modeling *saboteur* represent new edges of the circuit, as such their presence is considered during the generation of Kirchhoff's equations. The system of equations describing a conservative net inevitably contains an equation describing either the potential or the flow of the injected edge. Concerning manipulated parameters modeling *mutants*, the abstraction developed by this work allows mark and preserves parameters of the original design. The abstraction does not require a prohibitive amount of time to complete; thus, a designer can quickly change which parameters must be preserved during the process. We can also rely on the Waveform Relaxation (WR) interfaces shown in Section 3.7 to reduce the complexity of solving the system of equations if we are interested in preserving a considerable amount of parameters.

5.2.4 Building the analog simulation model

This section explains how the analog abstracted macro-blocks are recombined to create the C++ model ready to simulate.

Each abstracted macro-block is a C++ class containing: both fault-free and faulty behaviors, the procedures for switching between the two, and the procedures required to update the WR interfaces. Each C++ simulation model inherits from the interface class shown in Listing 5.4, which provides the two functions, *process* and *update_interfaces*. The former is called at each simulation step and receives as parameter the size of the simulation step. The latter function is called when all the processes have advanced by one simulation step, and retrieves the internal values required to update the WR interfaces. During a single simulation step, the simulator iterates through all the components and calls their *process* function, then it does

Listing 5.5: The *process* function of the inverter written in C++ instrumented with the switch statement for fault simulation.

```
 1 // Injection control variables.
 2 uint32_t instance_number = 0UL;
 3 uint32_t fault_number    = 0UL;
 4
 5 void Inverter::process(double ts)
 6 {
 7     // Switching point evaluation.
 8     x = b_a.pot;
 9     x_u = tanhsw(x, vtsh_u, smth_u);
10     x_d = tanhsw(x, vtsh_d, smth_d);
11     // Perform smooth switching.
12     c0 = x_u*a_vdd_cap_low + (1-x_u)*a_vdd_cap_high;
13     c1 = x_d*a_vss_cap_low + (1-x_d)*a_vss_cap_high;
14     c2 = x_u*vdd_q_cap_low + (1-x_u)*vdd_q_cap_high;
15     c3 = x_d*vss_q_cap_low + (1-x_d)*vss_q_cap_high;
16     r0 = x_u*vdd_q_res_low + (1-x_u)*vdd_q_res_high;
17     r1 = x_d*vss_q_res_low + (1-x_d)*vss_q_res_high;
18     // Components evaluation.
19     switch(((instance_number == _id) * fault_number))
20     {
21         default:
22         case 0: // Open-Circuit 1
23             b_a_vdd.pot = (*\itshape expr1*);
24             b_a_vdd.flw = (*\itshape expr2*);
25             ...
26         case 1: // Open-Circuit  2
27         case 2: // Short-Circuit 4
28             b_a_vdd.pot = (*\itshape expr3*);
29             b_a_vdd.flw = (*\itshape expr4*);
30             ...
31         case 3: // Short-Circuit 2
32             ...
33     }
34     // WRL waves evaluation.
35     b_wrl_a.pot = ...;
36     b_wrl_q.pot = ...;
37 }
```

the same with *update_interfaces*. This operation is repeated until all the components have achieved convergence before advancing the time.

Listing 5.5 shows the code ready for fault simulation produced by the proposed methodology applied to the inverter shown in Listing A.7. The first set of instructions performing states switching have a direct representation in C++, thus they are identical to the Verilog-AMS. The code after line 19 is the abstracted code of the inverter, which is explained more in details in the following paragraphs.

In this high-level description, each edge of the circuit is represented by the structure shown in Listing 5.6, which keeps track of its potential difference and flow (*i.e.*, the pair of variables *pot* and *flw* respectively). Each variable which appears in Listing 5.5 with the prefix *b_* is an edge of the circuit. During simulation these values are updated inside the *process* function by evaluating the arithmetical expression produced by the abstraction. For instance, potential difference and flow of the branch *b_a_vdd* are evaluated respectively at lines 23 and 24 of Listing 5.5. The voltage waves required by the WR interfaces are calculated inside this function in the same

Listing 5.6: High-level description of a circuit edge.

```
struct edge_t {
    double pot, flw;
};
```

way, as shown in Listing 5.5 at lines 35 and 36. In this particular case, the inverter required a WR interface connected to each port (*i.e.*, *a* and *q*). At the end of each simulation step, the *update_interfaces* function updates its WR interface and all the other interconnected WR interfaces with these waves values. The complete expressions are not reported in the listing, for the sake of readability they are replaced by: *expr1*, *expr2*, *etc*.

An analog fault can impact on different edges of the circuit and vice versa. Thus, the equations evaluating the potential and flow of such edges change based on the currently active fault. During the simulation, the model's behavior can be switched between fault-free and faulty through the use of a `switch` statement, based on the pair of global variables `instance_number` and `fault_number`. See line 19 of Listing 5.5 for example. These are the same variables used for the digital fault injection shown in Section 5.1. The first variable controls which instance of a component is currently injected and is compared against `_id`, which uniquely identifies a component instance. The second variable controls which fault is currently active. During the generation of these *switch* statements, if two or more cases have the same set of expressions they are collapsed into one. This happens when multiple faults have equivalent behavior, and as a consequence they produce the same set of equations. This optimization can drastically reduce the number of faults that to simulate.

As specified in Section 5.2.2 the behavior of *parametric faults* is modeled by a specific parameters configuration. In this particular case, the switching between different parametric faults is performed during the instantiation of the model, while the *process* function just implements the fault-free behavior. As a result, this configuration has no simulation overhead *w.r.t.* the fault-free version.

5.3 Network fault injection

Network fault models represent the wrong behavior of packet-based asynchronous data transmissions. In today's communications, information is delivered as *packets*, *i.e.*, sequences of bytes starting with a header bearing source and destination indication and other essential data used by the protocol to handle the message (*e.g.*, sequence number, payload type).

Packets may be subject to the following problems:

1. *Delay*: a packet may reach the destination later than expected;
2. *Drop Sender*: the packet is lost at the transmitter;
3. *Drop Receiver*: the packet is lost in the way to the receiver;
4. *Duplicate*: the packet is sent twice by the transmitter;

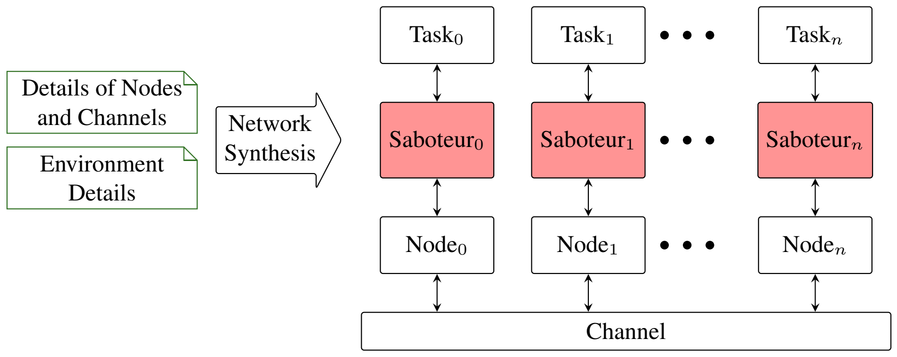


Fig. 5.7: SCNSL architecture with entities for fault injection.

5. *Bit-Flip*: bits of the byte sequence are flipped.

Transmission delay may be due to medium access contention, waiting time in the queues of intermediate systems (especially in case of congestions) and retransmission of lost packets if implemented by the protocol. Both network transmitter and receiver could fail to handle the packet. This kind of fault can be due to network congestions, a bug in the network library inside the software, or even bad or too low performances of the microprocessor. We decided considering both cases because in the latter case the packet is lost on the receiver side but in the meanwhile, it has occupied the channel. For similar reasons a packet which is supposedly lost, but it is not, could be retransmitted and thus leading to a duplicate packet. Generally, a bit-flip may be due to physical interferences on the channel, when the received signal is too weak because of distance or obstacles, and due to multi-path fading when the direct-path signal partially overlaps with its reflected copies.

Figure 5.7 shows the proposed architecture for the network-oriented fault simulation. The proposed injection architecture is based on an extension of the SystemC Network Simulation Library (SCNSL) [114] which provides the primitives to build and simulate network scenarios by using SystemC and C++ [115]. The *nodes* represent the embedded systems involved in the networked cyber-physical system; they interact together through a model of the communication *channel*. The *tasks* are the active entities which produce or consume packets; they are hosted by nodes. Network *saboteurs* are introduced between each task and the corresponding node by extending the *Communicator* interface provided by SCNSL. This results in the saboteur being triggered each time a packet transit between the task and the channel.

The proposed methodology is an extension of the network synthesis flow presented in Chapter 4. At the end of the synthesis flow, details of nodes, channels and the surrounding environment are used to configure the network saboteurs, *e.g.*, their injection probabilities. The mechanism of activation for network saboteurs is the same used for activating digital and analog faults. Each network saboteur is identified by an unique *id*, which is compared against the global variable *instance_number* to determine if it is currently active. While the variable *fault_number* controls the type of fault that the active saboteurs must inject.

Listing 5.7: Function used to set the *Delay* network fault.

```
// Define the type of the function used to generate delay values.
using std::function<double()> = delay_function_t;
// Store the object used to inject delays on transiting packets.
delay_function_t _delayFunction;
// Function used to set the saboteur delay function.
inline void setDelayFunction(delay_function_t fun) {
    _delayFunction = fun;
}
```

Listing 5.8: Example of network saboteur initialization through *lambdas*.

```
saboteur = new Protocols::MySaboteur("saboteur", 0);
saboteur->setDuplicateFunction([](){
    return 0.75;
});
saboteur->setDropReceiverFunction([](){
    return 0.55;
});
saboteur->setBernoulliBitFlipFunction([](){
    return 0.35;
});
saboteur->setDistanceBasedBitFlipFunction([](){
    return 0.25;
});
saboteur->setDropSenderFunction([](){
    return 0.15;
});
saboteur->setDelayFunction([](){
    return 0.15;
});
```

5.3.1 Implementation of saboteurs

Each network saboteur extends the *Communicator* interface of SCNSL, and contains a set of `std::function` object, one for each type of fault. The class `std::function` is a general-purpose polymorphic function which points to a callable target (*e.g.*, functions, lambda expressions, *etc.*). Before starting the simulation, these objects are used to store the functions controlling the injection of network faults.

Listing 5.7 shows the lines of code used for setting up the injection of delays. In this example, *delay_function_t* is a *type alias* declaration, it defines a *synonym* for the callable object which accepts no argument and returns a value of type *double*. This type is used inside the saboteur to store a copy of the callable object, namely *_delayFunction*. This object is set by using the *setDelayFunction* function, and it is used during the simulation to implement the desired injection policy. The same mechanism is used to set-up all the other types of network faults.

Listing 5.8 shows the initialization of a saboteur and its injection functions. During the simulation setup, the saboteur stores a function pointer for each fault model, which returns a value of type *double* representing either the fault rate (in case of packet drop, packet duplication, and bit flip) or the maximum delay. The use of this mechanism of "function pointers" instead of traditional parameters allows changing the saboteur behavior even during the simulation. For instance, the probability of a bit-flip could be related to a communication or an environmental condition which may change during simulation (*e.g.*, multi-path fading).

The injection functions return a value of type *double* between 0 and 1. Saboteurs related to packet loss or duplication are implemented as success/unsuccess Bernoulli stochastic processes which uses the *std::bernoulli_distribution* class of the standard library of C++ [1]. The delay saboteur instead, takes as input the maximum delay that can be applied to each packet and defers the transmission of the packet for a random delay uniformly distributed between 0 and such maximum value.

5.3.2 Bit-flip saboteur

There are two implementations of the bit-flip saboteur. The first uses the same Bernoulli approach as for packet loss but at bit level instead of packet level. Since this approach could lead to an excessive simulation overhead, we implemented an alternative bit-flip saboteur that randomly computes the distance between consecutive bit-flips according to the bit error rate. This approach reduces significantly the number of calls to the random number generator *for each bit*. Listing 5.9 sketches the main details of the code implementing such distance-based approach. It is called for each packet sent by the given node.

Let *bitstream* be the whole sequence of bits transmitted from the sender node to the receiver node if we ideally remove byte and packet boundaries. Given a bit error rate B, the *average distance* between two consecutive errors in the bitstream is $d = \frac{1}{B}$. Therefore, the random number generator is called *for each error* we want to generate to compute its distance from the previous error. For this purpose, the *RealRand* function generates the distance value based on a uniform real distribution from 1 to $2d$. Listing B.2 shows the implementation of the *RealRand* function. Once the actual distance is computed, the algorithm has to efficiently locate the bit to be changed either in a specific byte of the current packet or in one of the following packets. The `_nextError` variable stores the position of the next error in the bitstream according to the drawn distance value; its value should be preserved from one call to the other since the next error can be outside the current packet. The `_old_BER` variable is used to check if the desired bit error rate changes from one call to the other (in that case the `_nextError` variable should be recomputed).

[1] http://en.cppreference.com/w/cpp/numeric/random/bernoulli_distribution

Listing 5.9: Code for the distance-based bit-flip injection.

```
void MySaboteur::perform_bit_flip(Packet_t & p)
{
    // Retrieve the bit error rate.
    double new_BER = _bitErrorRateFunction();
    // If the error rate is lower than a given threshold
    // or higher than 1, completely skip the injection.
    if ((new_BER < epsilon) || (new_BER > 1)) {
        return;
    }
    // Retrieve the payload.
    byte_t * payload = nullptr;
    p->getPayload(payload);
    // Get the size of the payload.
    const size_t ps = p->getPayloadSizeInBytes();
    // Check if the bit error rate has changed.
    bool ber_changed = !is_equal(new_BER, _old_BER);
    // Get the maximum distance between each error.
    double distance = (1 / new_BER) - 1;
    // If the bit_error_rate has changed, update
    // the stored one, and re-evaluate the next-error.
    // This will be called also at the first execution
    // of the saboteur.
    if(ber_changed) {
        _old_BER = new_BER;
        _nextError = RealRand<double>(1, 2 * distance);
    }
    // While the next error is inside the payload.
    while (static_cast<size_t>(_nextError) < (ps * 8)) {
        // Get the byte and the bit.
        const size_t byte = size_t(_nextError) / 8;
        const size_t bit  = size_t(_nextError) % 8;
        // Corrupt the byte.
        toggle_bit<size_t>(payload[byte], bit);
        // Get the next error.
        _nextError += RealRand<double>(1, 2 * distance);
    }
    // Offset the next error based on the bytes of the
    // current payload.
    _nextError -= (ps * 8);
}
```

6
Holistic functional safety evaluation

This chapter has the purposes of evaluating both the correctness and efficiency of the proposed fault models and show the application of the entire methodology to a realistic Networked Cyber-Physical System (NCPS). To assess the effectiveness of the presented methodology for the functional safety of a networked cyber-physical system a complex case study has been developed. To exemplify the potentialities of the proposed approach and of the holistic platform, the diagnostic coverage metric is evaluated for each of its safety mechanism.

6.1 Case study

The proposed case study recalls a today's scenario for smart manufacturing in which machines of a production line are connected to a remote system that monitors their behavior to predict possible failures and plan maintenance without out-of-order intervals. The scenario is depicted in Figure 6.1 and consists of three wireless sensor nodes attached to production machines that send the vibration information retrieved by an accelerometer to a computation node. Each node is equipped with a *MOS Technology 6502* 8-bit microprocessor, 1 MB RAM, 8 KB ROM, a 3-axis accelerometer, and a wireless interface. Once a node is powered on, a minimal bare metal kernel is loaded from the ROM into the RAM and executed. One of the four wireless nodes has sensibly higher capabilities and acts as data collector. A shared wireless channel is established between each remote node and the data collector. The whole system can be considered a networked cyber-physical system and Figure 6.1 shows its architecture with reference to the three modeling domains, *i.e.*, digital, analog and network.

6.2 Injection techniques evaluation

The entire design has been abstracted and injected with the techniques described in the previous chapters. For the *digital* part, two C++ versions of the injected design

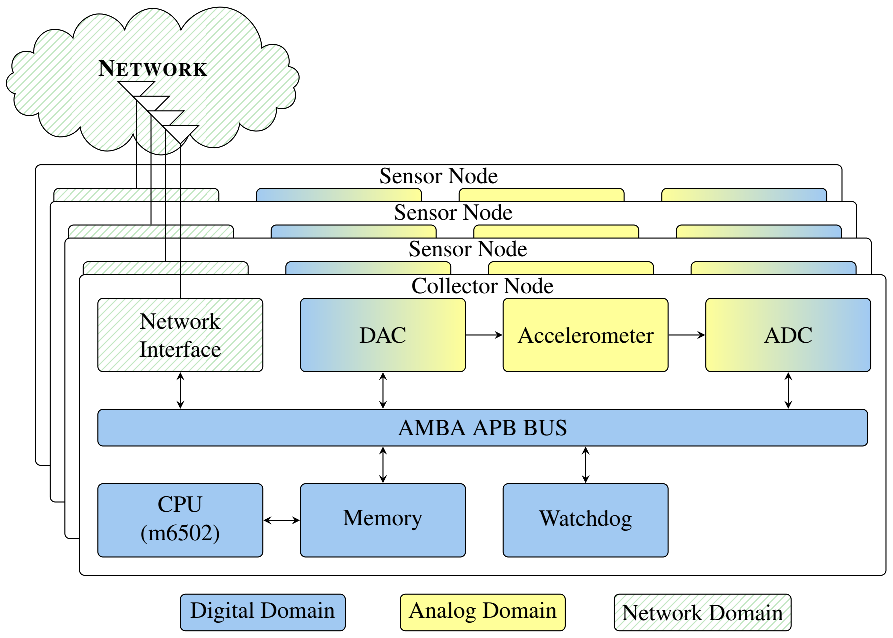


Fig. 6.1: Architecture of the networked cyber-physical system.

are generated, one for the instrumentation with functions and one with bitwise operators. The total number of digital faults injected in every version is 4151, among 536 mutation locations found. For what concerns the analog component, the accelerometer has been injected using controlled current sources which model a current spike (*e.g.*, alpha particles hitting sensible nodes). In this case, the total number of injected analog faults is 50. The network part instead, counts a total of 36 faults. There are 6 network fault models, *i.e.*, 12 faults for each bidirectional connection. The total number of faults is 4237.

6.2.1 Network saboteurs

Three tests have been designed to evaluate the network saboteurs.

The *first test* aims at evaluating the actual behavior of the implementation of the proposed network saboteurs. Injection data have been retrieved in a simulation involving the transmission of 10,000 packets. The initialization parameters of Listing 5.8 have been used. For instance, the probability for a packet to be lost by the sender is 0.15, the probability for a packet to be lost by the receiver is 0.55, the maximum applicable delay is 0.15 seconds and so forth. Results are shown in Figure 6.2. Resulting fault rate values are reported with reference to the vertical scale on the left while resulting delay values (*i.e.*, pentagon marks) refer to the vertical scale on the right. The trend lines show that the mean fault rate values match the fault probability values used during initialization and the mean delay is about 0.075 seconds, exactly

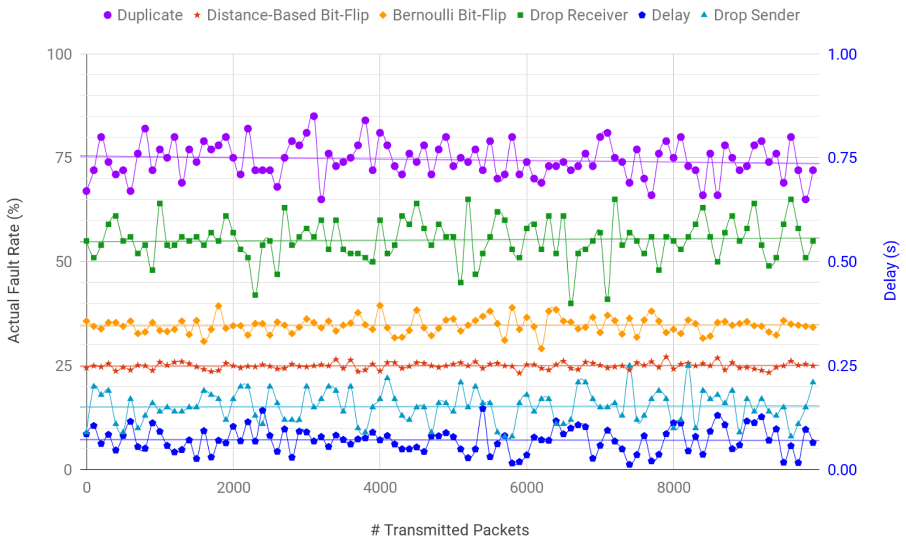


Fig. 6.2: Actual injection behavior for various network saboteurs.

half of the maximum delay used during initialization. Furthermore, the comparison between Distance-based Bit-Flip saboteur with respect to the traditional Bernoulli Bit-Flip saboteur shows that the implementation of the former is more stable than the latter since the actual bit errors better match the desired bit error rate with smaller deviation.

The *second test* aims at evaluating the efficiency of the packet-based network saboteurs, *i.e.*, those that alter the timed sequence of transmitted or received packets. The impact of this kind of saboteur on simulation time has two contributions, *i.e.*, the overhead of saboteur computations and the effect of the fault injection on network simulation (*e.g.*, packet duplication increases the number of packets to be handled by the simulator). Since we are interested in evaluating the first contribution, in the experiments we set fault rate to 0% for drop and duplication saboteur and to 0 seconds for delay saboteur. Table 6.1 shows CPU time and overhead for each saboteur with

Table 6.1: Simulation overhead of packet-based network saboteurs.

Fault Model	Time (s)	Overhead (%)
Fault-Free	3.16	reference
Delay	5.26	67
Drop Sender	5.38	70
Drop Receiver	5.57	76
Duplicate	5.72	81

respect to non-instrumented model simulation. As expected, the simulation overhead is quite the same and equal to about 75%.

The *third test* concerns specifically the implementation of bit-flip injection. Figure 6.3 shows the simulation time of the two implemented techniques as a function of the bit error rate. Non-instrumented model simulation is also reported as a reference. As expected, the traditional Bernoulli bit-flip technique has a constant overhead since it draws the chance of injection for each bit of the packet. At low bit error rate values, the Distance-based bit-flip method outperforms the traditional approach because it processes only the bits that need to be changed to obtain the expected bit error rate. The overhead increases with the bit error rate to reach the one of Bernoulli approach at 100% fault rate. Considering that 50% bit error rate can be regarded as a reasonable upper bound in most of real scenarios; we can say that Distance-based bit-flip technique adds a maximum overhead of about 53% which is 23% less than Bernoulli bit-flip technique.

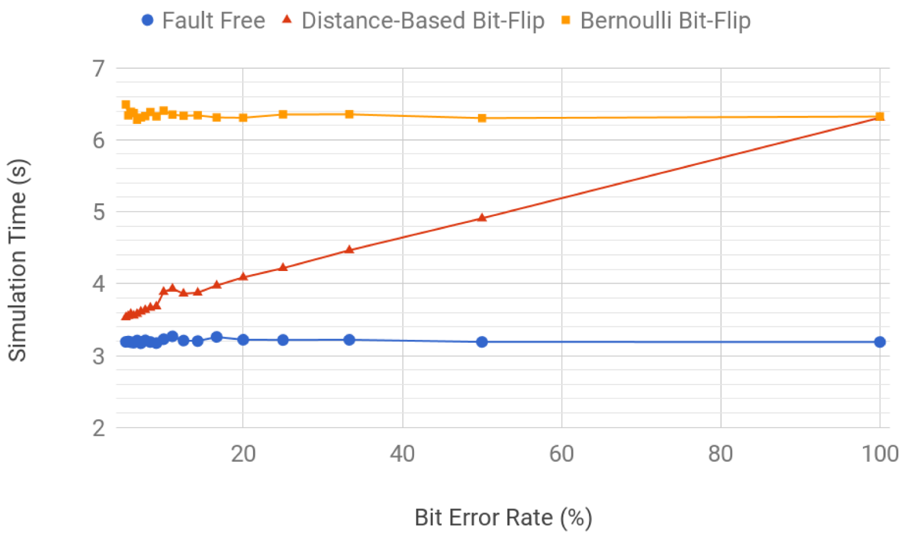


Fig. 6.3: Simulation overhead of the Distance-based Bit-Flip saboteur compared to Bernoulli Bit-Flip saboteur as a function of the injected bit error rate.

6.2.2 Digital mutants

Two tests have been designed to evaluate the efficiency of digital mutants.

The *first test* evaluates the overhead due to the presence of mutants inside the high-level C++ code. Two versions of the M6502 have been generated, each one instrumented with a different injection methodology taken from Section 5.1, *i.e.*, functions, and bitwise operators. Since the purpose is evaluating the injection overhead, no fault is active during the simulation. The number of digital processes in

Table 6.2: Simulation overhead evaluation of digital mutants.

Code Version	Time (s)	Overhead (%)
Fault Free (C++)	3.241	reference
Functions	5.287	63
Bitwise Operators	3.761	16

Table 6.3: Simulation times to perform digital fault analysis.

Code Version	Time			Speedup
Verilog (RTL)	46h	26m	31.000s	1.00
Functions	2h	57m	38.887s	15.68
Bitwise operators	2h	6m	28.854s	22.03

the M6502 abstract model is 77. Table 6.2 reports the mean execution times over 100 runs compared with the simulation time of the fault-free version of the microprocessor. As expected, using bitwise operators and compile-time optimized constructs (macros and inline functions) provide higher performance than using the approach with function calls. In that it introduces a negligible degradation with respect to the non-instrumented model.

The *second test* evaluates the efficiency of the C++ digital fault injection *w.r.t.* the original Verilog injection at Register-Transfer Level (RTL). In this scenario, the execution of the fault-free platform is compared with the faulty one to determine the presence of faults. The faulty design description executes the workload once for each fault and enables one mutant per time. During the execution the values of output ports are compared after each clock cycle. If a difference is found, the execution of the workload for that mutant is stopped and the fault is considered as tested (or killed). If the execution ends with no difference on the output ports, the fault is regarded as untested (or alive). The execution times are reported in Table 6.3. The values highlight that the approach with bitwise operators performs better than the approach with functions.

6.2.3 Analog faults

Two tests have been designed to evaluate the analog fault injection technique.

The *first test* aims at evaluating the simulation efficiency and instrumentation overhead of the methodology on a set of nine benchmarks taken from Chapter 2. Table 6.4 reports the simulation time required by Questa-ADMS and the proposed framework for simulating a fault-free description, a faulty version with no active fault and a simulation of all the injected faults. For the faulty version with no active fault we have reported the overhead introduced by the manipulation, both in the original code and in the abstracted one. The results report the simulation time required to

Table 6.4: Execution times of fault-free, single injected and all faults benchmarks.

	Verilog-AMS				Testing Framework – C++					
	Fault Free	No Active Fault		All Faults	Fault Free	No Active Fault			All Faults	
Benchmark	time (s)	time (s)	over (%)	time (s)	time (s)	time (s)	over (%)	speed-up (x)	time (s)	speed-up (x)
RC1	5,046.72	5,487.12	8.73	5,523.44	4.87	5.91	21.36	**928.45**	5.87	**940.96**
IN2	5,028.54	5,828.67	15.91	11,561.44	4.46	5.16	15.70	**1,129.59**	10.12	**1,142.43**
PIFilter	5,081.49	6,133.54	20.70	12,340.74	5.10	6.31	23.73	**972.03**	12.91	**955.91**
IN3	5,312.51	6,239.96	17.46	18,713.28	4.89	5.75	17.59	**1,085.21**	17.21	**1,087.35**
Op-Amplifier	5,741.43	6,978.23	21.54	21,159.44	4.93	5.83	18.26	**1,196.95**	17.81	**1,188.06**
RC5	5,763.95	7,408.76	28.54	37,168.75	6.56	9.28	41.46	**798.36**	46.78	**794.54**
RC10	6,704.85	9,588.87	43.01	96,220.30	13.12	23.33	77.82	**411.01**	236.83	**406.28**
RC20	8,512.29	17,453.71	105.04	330,286.14	54.31	101.78	87.41	**171.48**	1,879.30	**175.75**
Accelerometer	11,917.28	18,517.34	55.38	462,407.91	47.72	67.95	42.39	**272.51**	1,584.71	**291.79**

execute *1 second* of simulated time. The executions were not halted if a fault was detected during the simulation with all the faults.

The notable results obtained with the testing framework over the different benchmarks emphasize the effectiveness of the proposed flow. The generated framework is able to achieve from two to three orders of magnitude of speed-up, thus proving to be a valuable solution for the dependability evaluation of an analog device.

The *second test* evaluates the efficiency of the analog fault injection technique on the *clock generator* test case presented in Section 3.9.4. Also in this case we applied the abstraction and injection methodology to the behavioral-level macro-blocks of the *clock generator*. Table 6.5 reports the time required to simulate $1us$ in different scenarios at all the abstraction levels. The first three scenarios refer to the fault-free non-instrumented version of the circuit at transistor, behavioral, and functional level. The fourth scenario shows the simulation time for the C++ instrumented version with no active fault, while the last scenario reports the time required to iteratively simulate all the injected faults.

Table 6.5: Simulation time required to perform a complete fault simulation of the clock generator for a total of thirty faults.

Abstraction Level	Scenario	Simulation Time (s)
Transistor	Fault free	106.70
Behavioral	Fault free	24.48
Functional	Fault free	13.71
Functional	Instrumented	13.96
Functional	All faults	28485.16

The number of mutated components is 352, with a total of 2122 faults injected with the proposed methodology. The simulation overhead introduced by the injection infrastructure is around 2% for the presented case study. This value is the result of the comparison between the simulation time for the fault-free and the instrumented C++ version shown in Table 6.5. Thanks to this negligible overhead the gain with respect to both the behavioral and the transistor-level version is preserved. This work does not directly address the problem of simulating parametric faults, however, the remarkable simulation performances can be used to test a great number of deviations of designs parameters.

6.3 Holistic functional safety evaluation

The purpose of this section is to show an example scenario where the proposed injection methodology could be used effectively. It is a typical example of diagnostic coverage metric evaluation, where a designer is interested in evaluating the effectiveness of safety mechanisms.
The safety mechanisms implemented for this architecture are the following:

- *Watchdog* – It checks if the CPU executes the workload in a fixed number of clock cycles. It consists of a counter with two input signals: `en` that enables the counter and `reset` that sets the counter to 0. An `expired` signal is raised by the watchdog when the counter reaches the fixed number of clock cycles.
- *Software (Power On Self Tests (POST))* – It checks if the CPU, ALU and registers are functioning correctly by executing a series of operations (*e.g.*, function calls, arithmetical operations, control statements). The outcome of each operation is compared with a reference value known a priori. If the POST sequence succeeds, then the variable `test_cpu` is set to 1, otherwise it is set to 0. The last instruction of the software, before the return statement, is an assignment of the value 1 to the variable `end_test`.
- *Software (Network)* – It checks if the connection between the central and remote nodes has been established and works correctly. It is executed after the POST. The central node sends a predefined sequence of values to the remote ones. A remote node receives the sequence, sums the values and sends the result to the central node. The central node checks if the received sums are correct. As such, the central node is able to determine malfunctioning connections. All the nodes set the variable `test_network` to `1` if the test is successful.
- *Software (Analog)* – Through an A/D interface, the software checks whether the accelerometer is behaving correctly or not. This test is designed as a Mixed-Signal Built-In Self-Test (MS-BIST). First, the accelerometer is isolated from the external physical environment. Then, a series of predefined stimuli (stored inside the ROM) are provided to it. The response of the accelerometer is checked for the presence of faults (*e.g.*, overshooting, ringing, noise) by means of an analog-to-digital conversion and a specific software code. If the test succeeds, then the variable `test_analog` is set to 1.

The watchdog is written in RTL Verilog and it has been abstracted along with the rest of the platform to the functional-level. The hardware failure mode is detected if and only if the watchdog raises the signal `expired`. In order to detect faults activated by using POST testbench checks the values written on the RAM. In details, if the `end_test` value is written and the `test_cpu` variable has value 0, then the software failure mode is detected. Otherwise if the `test_cpu` variable has value 1 at the end of the execution, then the injected fault did not affect that failure mode.

To detect network faults, the testbench checks the value of `test_network`. Similarly to the digital failure mode, if `end_test` value is written and the `test_network` variable has value 0, then the network fault is considered as detected.

Analog faults are detected whenever the values of the accelerometer differ from the pre-calculated ones. If `end_test` value is written and the `test_analog` variable has value 0, then the analog fault is considered as detected.

Before analyzing the diagnostic coverage evaluation, and the simulation results, let us clarify the meaning and purpose of a safety mechanism in the context of a holistic platform. Let us consider the POST, which is a safety mechanisms targeting specifically faults inside the CPU. By construction the POST is executed by using values stored inside the ROM and relies only on components belonging to the CPU. As a consequence, the faults that are considered potentially *detectable* by this mechanism are only those inside the CPU. Instead, if the POST executes operations with the support of an external Floating-Point Unit (FPU) (*e.g.*, for efficiency reasons), the faults that are considered potentially *detectable* by this mechanism are those inside the CPU plus those in the FPU. The same reasoning can be applied to the scenario with the MS-BIST, where the CPU interacts with the accelerometer to test its correctness. Also in this case the mechanism could potentially *detect* faults inside both the CPU and the accelerometer, not only those inside the latter. So the nature of a safety mechanisms is related to the components which influence its evaluation, *i.e.*, functional safety metrics.

Let us now introduce the set of variables we require for the diagnostic coverage evaluation as formalized in the standard IEC 61508 [116]. For the purpose of this experiment we focus on the total number of:

$$\begin{aligned}\lambda_{dd} &= \textit{“dangerous” detectable failures}\\ \lambda_{du} &= \textit{“dangerous” undetectable failures}\\ \lambda_{d} &= \textit{“dangerous” failures}\end{aligned}$$

where λ_d is the sum of the first two:

$$\lambda_d = \lambda_{dd} + \lambda_{du}$$

The original IEC 61508 document also defines the number of “safe” failures represented by the variable λ_s, which are not in the scope of this work. The formula used to evaluate the *diagnostic coverage* is defined as a percentage as follows:

$$DC\% = \frac{\lambda_{dd}}{\lambda_d} \times 100$$

In the proposed case study there are a total of four safety mechanisms making up our entire safety system. The following set of variables identify the number of faults flagged by each safety mechanism (*i.e.*, the detectable ones λ_{dd}):

$$
\begin{aligned}
\lambda_{WD} &= \textit{flagged by Watchdog} = 1641 \\
\lambda_{POST} &= \textit{flagged by POST} = 22 \\
\lambda_{MSBIST} &= \textit{flagged by MSBIST} = 46 \\
\lambda_{NET} &= \textit{flagged by NET} = 14
\end{aligned}
$$

Each one of these safety mechanisms covers one or more components or aspects of the holistic platforms. Let us now define a set of variables identifying the total number of "dangerous" failures:

$$
\begin{aligned}
\lambda_{d_{cpu}} &= \textit{in the CPU} = 4151 \\
\lambda_{d_{acc}} &= \textit{in the accelrometer} = 50 \\
\lambda_{d_{net}} &= \textit{in the network} = 36
\end{aligned}
$$

We can evaluate the *diagnostic coverage metrics* for each safety mechanism as follows:

$$
\begin{aligned}
DC_{WD} &= \frac{\lambda_{WD}}{\lambda_{d_{cpu}}} \times 100 = \frac{1641}{4151} \times 100 = 39.53\% \\
DC_{POST} &= \frac{\lambda_{POST}}{\lambda_{d_{cpu}}} \times 100 = \frac{22}{4151} \times 100 = 0.53\% \\
DC_{MSBIST} &= \frac{\lambda_{MSBIST}}{\lambda_{d_{cpu}} + \lambda_{d_{acc}}} \times 100 = \frac{46}{4151+50} \times 100 = 1.09\% \\
DC_{NET} &= \frac{\lambda_{NET}}{\lambda_{d_{cpu}} + \lambda_{d_{net}}} \times 100 = \frac{14}{4151+36} \times 100 = 0.33\%
\end{aligned}
$$

Thanks to the holistic nature of the simulation scenario, we can use as common denominator the number of faults with which the safety mechanism comes into contact. Table 6.6 reports the comparison between the diagnostic coverage metrics evaluated with the components in isolation and inside the holistic platform.

Table 6.6: Diagnostic coverage metric evaluated with the component in isolation and inside the holistic platform.

Safety Mechanism	Isolation	Holistic Platform
DC_{WD}	39.53%	39.53%
DC_{POST}	0.53%	0.53%
DC_{MSBIST}	92.00%	1.09%
DC_{NET}	38.88%	0.33%

The diagnostic coverage assessed using the holistic platform takes into account all the faults that influence its evaluation, as a consequence, it is lower than the one evaluated with the component in isolation. This aspect will be further highlighted in the following section, where a safety mechanism is shown that can cover multiple components belonging to different domains.

The time required to perform the entire diagnostic coverage evaluation is 5 *hours*, 15 *minutes* and 74 *seconds*. The execution time is higher than the ones reported in Table 6.3 because the simulation is not dropped when a failure mode is detected and the workload is executed entirely for each fault.

6.3.1 Multi-domains safety assessment

The last experiment wants to highlight the effectiveness of the proposed methodology in improving functional safety evaluation by considering multiple domains. This scenario uses a modified version of the case study presented in Section 6.3. A new safety mechanism is introduced which merges analog and network tests. The remote nodes do not store the stimuli for the accelerometer inside the ROM, but instead, they receive them through the network connection from the central node. After POST the remote nodes receive the stimuli and apply them to the accelerometer. Then, they read the values from the accelerometer and return the samples to the central node which checks the correct behavior of the analog component. Centralized checking of all the analog components could be an interesting solution for decreasing the software and hardware complexity of remote nodes and thus their cost.

With this scenario, a network fault could lead not only to a faulty or missing network connection but also to wrong analog test sequences which can compromise the functional safety evaluation. Simulating this scenario with the holistic platform showed that a series of bit-flip faults which were not detected by the network tests led to a wrong set of stimuli for the accelerometer. Consequently, the analog safety mechanism detected anomalies in the accelerometer's behavior and raised the error flag. The number of faults flagged by this multi-domain safety mechanism is identified by the variable:

$$\lambda_{MD} = \textit{flagged by multi-domain mechanism} = 81$$

which is used to evaluate the diagnostic coverage metric as follows:

$$DC_{MD} = \frac{\lambda_{MD}}{\lambda_{d_{cpu}} + \lambda_{d_{net}} + \lambda_{d_{acc}}} \times 100 = \frac{81}{4151 + 50 + 36} \times 100 = 1.91\%$$

The purpose of this multi-domain safety mechanism is detecting both network and analog faults, with a greater chance *w.r.t.* mechanisms targeting only one of the two domains. However, even this multi-domain approach has some drawbacks. An anomaly could be caused by a failure either in the network or analog domain, and consequently, it could be hard to distinguish between these two cases. Furthermore, failures in both domains can mask each other and may not be detected at all. The proposed holistic platform allows exploring and in some cases addressing these kind of issues, which otherwise would require using more inefficient approaches.

7

Conclusion and suggestions for future research

To conclude this thesis, we summarize the methodologies which lead to the holistic approach to functional safety for networked cyber-physical systems, and suggest directions of future research that builds on the proposed approach.

7.1 Summary of the proposed approach

The study was set out to explore the problems related to the **functional safety** evaluation for Networked Cyber-Physical System (NCPS). A NCPS is a system of systems which cooperate and interact together to implement a global application or achieve a common goal. Each separate system is a sophisticated device with computation, communication, sensing, and actuation aspects. So we can summarize that a NCPS is a *highly heterogeneous* device comprising analog and digital hardware, and network capabilities. It is clear that ensuring the functional safety of such systems is becoming an increasingly complex task.

The thesis starts from the root of all problems, *i.e.*, the efficient simulation of analog components. To deal with this problem, it introduces the *analog abstraction* methodology, which aims at moving the complexity of simulating analog components from simulation-time to generation-time. It proposes a mixed-signal scheduling approach which aims at reducing the overhead due to the synchronization between analog and digital processes. It shows that the proposed technique can sensibly reduce this overhead compared to commercial tools. Then, it introduces an approach to transform transistor-level descriptions into state-dependent small-signal behavioral models. This transformation allows applying the abstraction also to these elaborate descriptions, which are commonly found in the industry. The experimental results show that this manipulation can speed up the simulation of transistor-level description. However, the approach is still in its infancy and there is still work that has to be done to improve the speed up of this models. In the conclusion of this first part, the thesis introduces a set of interfaces, which rely on lossless transmission-line theory. These interfaces allow splitting conservative circuits which as a consequence reduces the complexity of solving equations systems. Furthermore, this interfaces allows the composition of already abstracted circuits, which until now was not feasible. To conclude, this first part of the this allows generating homogeneous Cyber-Physical System (CPS), efficiently combining analog and digital components inside the same description, written in C++. The purpose of this set of techniques is to improve the simulation efficiency and ease the steps that compose the design of a complex smart device.

The second part of the thesis faces another problem, expanding the standard design flow for embedded systems to cover the communication aspects that so characterize every modern smart device. This newly extended flow takes the name of *Communication-aware Design Flow* and relies on a formal description of the environment, distributed application, the tasks implementing the application, and the data flowing between the tasks. This formal description takes the name of *Communication-aware Problem Formulation*. The thesis builds upon this new formalism an automatic flow described as a Mixed Integer Linear Programming (MILP) problem, which generates optimal infrastructures for networked cyber-physical systems. A study of complexity and scalability shown that the current approach belongs to the family of NP-hard problems. Nevertheless, the main objective of this flow and the proposed formalism is to pose the first cornerstone upon which we start building future synthesis techniques. Experimental results shown the applicability of the flow in two real case studies, an in-building network and a wide urban area.

The final part of the thesis exploits the previous methodologies for improving the fault modeling and simulation in all the domains of a NCPS, *i.e.*, analog, digital, and network. It first extends the abstraction methodology to inject faults in analog descriptions automatically. It provides a taxonomy of analog fault models considered by this methodology, and for each one, it describes how they are injected and at which level of abstraction they can be used. Then, it proposes an efficient digital fault injection methodology with a sensibly low overhead introduced by the faulty code. This second injection method takes advantage of the compile-time optimizations enabled by C++ language and compilers. The third methodology is built upon the network synthesis flow and proposes an approach for the injection of network faults through saboteurs acting as protocol layers. In this case, saboteurs configurations are automatically generated based on the information provided in the communication-aware problem formulation.

This last methodology allows building a holistic simulation platform addressing the functional safety analysis for networked cyber-physical systems. The experimental results shows that the three injection techniques combined with abstraction methodologies can sensibly improve fault simulation efficiency. Thanks to the holistic nature of this platform, designers can analyze the correlation between safety mechanisms and faults spanning across all the domains of networked cyber-phyisical systems.

7.2 Directions for future research

New opportunities arise from the work of this thesis, some born thanks to limitations of the proposed techniques, some others because of the curiosity and eagerness to improve and explore. Follows a list of future research linked to this work, some of which we are already under study:

- The process used to generate the behavioral models starting from transistor-level descriptions, presented in Section 3.6, is not currently automated. We have formalized the steps required to achieve this goal, and automatized some of this steps. Our preliminary study shows that we can extend this process to more complex blocks comprising hundreds of transistors, and reduce them to these state-dependent small-signal models.
- The waveform relaxation interface presented in Chapter 3 are a stable approach. However, more studies are required to improve the convergence of this interfaces, and therefore the ability to achieve accurate behaviors with less iterations.
- The network synthesis and the formalisms presented in Chapter 4, has mentioned, are just the beginning. New activities are already studying how to combine the MILP formulation with heuristic approaches to sensibly reduce the complexity. Furthermore, the flexibility of the formulation inspired new extensions that better capture the aspects of devices for Industry 4.0.
- The experimental results and the ability to build holistic platforms opens new opportunities in fields that are not directly dealt in this thesis. One of the other uses that we are exploring is the efficient development of embedded software inside of smart platforms (*i.e.*, from the energy consumption point of view, the execution efficiency, ability to debug, *etc.*). Furthermore, there are several questions regarding the holistic platform that we want to answer in the near future:
  1. Determine if the safety mechanisms designed with the aid of the holistic platform are more effective than those designed in isolation.
  2. Opens to new speculations, *i.e.*, to determine if there are cases in which the correct behavior of many different components, then generates an incorrect behavior when they are combined.
  3. If it is true that correct behaviors in isolation, if put together generate something incorrect, we want to understand if it is possible to design safety mechanisms that can protect us from such worst cases.

Summary of the proposed innovative contributions

This chapter reports the innovative contributions to the *State of the Art* by the work proposed in this thesis. The articles are grouped in three main subjects as follows:

- the first group is the *analog mixed-signal multi-domain abstraction*, which comprises both Chapter 2 and Chapter 3;
- the second group contains articles related to the *synthesis for Networked Cyber-Physical Systems*, which refers to the methodology shown in Chapter 4;
- and the final group contains the articles related to the *fault modeling and simulation* which are linked to the content of Chapter 5.

The contributions inside each group are in chronological ordered.

Analog mixed-signal multi-domain abstraction

1. **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 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1586–1591, 2016, doi: 10.3850/9783981537079_0376.
2. **E. Fraccaroli**, M. Lora, F. Fummi, and P. Montuschi. "*A fast simulation environment for smart systems validation in presence of electromagnetic interferences.*" in International Conference on Electromagnetics in Advanced Applications (ICEAA), pp. 740–743, 2016, doi: 10.1109/ICEAA.2016.7731505.
3. M. Lora, **E. Fraccaroli**, and F. Fummi. "*Virtual prototyping of smart systems through automatic abstraction and mixed-signal scheduling*". in Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 232–237, 2017, doi: 10.1109/ASPDAC.2017.7858325.
4. **E. Fraccaroli**, M. Lora, and F. Fummi. "*Automatic abstraction of multi-discipline analog models for efficient functional simulation.*" in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 662–665, 2017, doi: 10.23919/DATE.2017.7927072.

5. M. Lora, S. Vinco, **E. Fraccaroli**, D. Quaglia, and F. Fummi "*Analog Models Manipulation for Effective Integration in Smart System Virtual Platforms.*" in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 37, no. 2, pp. 378–391, 2017, doi: 10.1109/TCAD.2017.2705129.
6. S. Centomo, M. Panato, **E. Fraccaroli**, and F. Fummi. "*From Multi-Level to Abstract-Based Simulation of a Production Line.*" **accepted to be published** in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1–6, 2019.
7. **E. Fraccaroli**, M. Lora, and F. Fummi "*Automatic Generation of Analog-Mixed Signal Multi-Discipline Virtual Platforms.*" **under submission** to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

Synthesis for Networked Cyber-Physical Systems

8. C. Barnes, J. Cottin, D. Quaglia, **E. Fraccaroli**, A. Pegatoquet, F. Verdier, and S. Angeleri. "*Network-aware virtual platform for the verification of embedded software for communications.*" in Euromicro Conference on Digital System Design (DSD), pp. 518–525, 2015, doi: 10.1109/DSD.2015.110.
9. **Enrico Fraccaroli**, Francesco Stefanni, Romeo Rizzi, Davide Quaglia, and Franco Fummi. "*Network Synthesis for Distributed Embedded Systems.*" in IEEE Transactions on Computers, vol. 67, no. 9, pp. 1315–1330, 2018, doi: 10.1109/TC.2018.2812797.

Fault modeling and simulation

10. **E. Fraccaroli** and F. Fummi. "*Analog fault testing through abstraction.*" in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, pp. 270–273, doi: 10.23919/DATE.2017.7926996.
11. **E. Fraccaroli**, L. Piccolboni, and F. Fummi. "*A homogeneous framework for AMS languages instrumentation, abstraction and simulation.*" in IEEE European Test Symposium (ETS), 2017, pp. 1–2, doi: 10.1109/ETS.2017.7968212.
12. **E. Fraccaroli**, F. Stefanni, F. Fummi, and M. Zwolinski. "*Fault Analysis in Analog Circuits through Language Manipulation and Abstraction.*" in Forum on specification & Design Languages (FDL), 2017, pp. 1–7, doi: 10.1109/FDL.2017.8303890.
13. **E. Fraccaroli**, D. Quaglia, and F. Fummi. "*Simulation-based holistic functional safety assessment for networked cyber-physical systems.*" in Forum on specification & Design Languages (FDL), 2018, pp. 5–16, doi: 10.1109/FDL.2018.8524050.
14. **E. Fraccaroli** and D. Quaglia. "*Efficient simulation of faults in networked cyber-physical systems.*" in Design of Circuits and Integrated Systems (DCIS), 2018, pp. 1–6.

15. R. Gillon, **E. Fraccaroli**, and F. Fummi. “*An Efficient Analog Fault-injection Flow Harnessing the Power of Abstraction.*” **accepted to be published** in Design and Verification Conference (DVCon) and Exhibition United States (US), pp. 1–6, 2019.
16. **E. Fraccaroli**, G. Renaud, and F. Fummi “*Physical-Aware Functional-Level Fault Simulation for Safety Analysis.*” **under submission** to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

References

1. L. Scheffer, L. Lavagno, and G. Martin, *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC Taylor & Francis, 2006.
2. S. Vinco and C. Pilato, "Editorial: Special Issue on Innovative Design Methods for Smart Embedded Systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 2, 2016.
3. S. G. Vijay K. Varadan, K. J. Vinoy, *Smart material systems and MEMS: design and development strategies*. John Wiley and Sons, 2006.
4. 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*, 2015, pp. 1–6.
5. 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 the IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–4.
6. 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*. Springer, 2016, ch. 3, pp. 23–54.
7. Imperas Software, "Ovp - open virtual platforms," . [Online]. Available: http://www.ovpworld.org
8. Cadence, "Virtual System Platform." [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/virtual-system-platform.html
9. Synopsys, "Platform Architect." [Online]. Available: https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html
10. Accellera, "Verilog-AMS Language Reference Manual," . [Online]. Available: https://www.accellera.org/downloads/standards/v-ams
11. F. Pêcheux, C. Lallement, and A. Vachoux, "VHDL-AMS and Verilog-AMS as alternative hardware description languages for efficient modeling of multidiscipline systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 2, pp. 204–225, feb 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1386377
12. Accellera, "SystemC-AMS Language Reference Manual." [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/SystemC_AMS_2_0_LRM.pdf

13. L. W. Nagel and D. O. Pederson, *SPICE: Simulation program with integrated circuit emphasis*. Electronics Research Laboratory, College of Engineering, University of California, 1973.
14. S. Vinco, M. Lora, and M. Zwolinski, "Conservative Behavioural Modelling in SystemC-AMS," in *Proc. of ECSI/IEEE Forum on Specification & Design Languages 2015 (FDL 15)*, 2015, pp. 1–8.
15. Mentor Graphics, "Questa Advanced Simulator." [Online]. Available: https://www.mentor.com/products/fv/questa/
16. M. Alassir, J. Denoulet, O. Romain, and P. Garda, "Modeling I2C Communication Between SoCs with SystemC-AMS," in *Proc. of IEEE ISIE 2007*, 2007, pp. 1412–1417.
17. 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*. IEEE, 2014, pp. 1–6.
18. F. Cenni, S. Scotti, and E. Simeu, "Behavioral modeling of a CMOS video sensor platform using SystemC AMS/TLM," in *Proc. of IEEE Forum on Specification and Design Languages (FDL), 2011*, 2011, pp. 1–6.
19. 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*, 2012, pp. 104–109.
20. 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 the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015, pp. 725–730.
21. 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 the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 250–255.
22. H. Aridhi, M. H. Zaki, and S. Tahar, "Towards improving simulation of analog circuits using model order reduction," in *Proc. of the IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 1337–1342.
23. A. Odabasioglu, M. Celik, and L. T. Pileggi, "PRIMA: Passive reduced-order interconnect macromodeling algorithm," in *Proc. of IEEE/ACM DAC 1997*, 1997, pp. 58–65.
24. H. Liu, L. Daniel, and N. Wong, "Model reduction and simulation of nonlinear circuits via tensor decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 34, no. 7, pp. 1059–1069, 2015.
25. P. Li and L. T. Pileggi, "Compact reduced-order modeling of weakly nonlinear analog and RF circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 24, no. 2, pp. 184–203, 2005.
26. P. Benner, "Solving large-scale control problems," *IEEE Control Systems*, vol. 24, no. 1, pp. 44–59, 2004.
27. 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*, vol. 16, no. 8, pp. 1024–1039, 2008.
28. C. Bauer, A. Frink, and R. Kreckel, "Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language," *Elsevier JSC*, vol. 33, no. 1, pp. 1 – 12, 2002.
29. Coventor, Inc., "MEMS+: MEMS Simulation Software." [Online]. Available: https://www.coventor.com/mems-solutions/products/mems-plus-overview/

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*, 2012, pp. 1–6.
31. P. Feldmann and R. Rohrer, "Proof of the number of independent Kirchhoff equations in an electrical circuit," *IEEE Transactions on Circuits and Systems (TCAS)*, vol. 38, no. 7, pp. 681–684, 1991.
32. V. Belevitch, "Summary of the History of Circuit Theory," *Proc. of the IRE*, vol. 50, no. 5, pp. 848–855, 1962.
33. N. Bombieri, M. Ferrari, F. Fummi *et al.*, "HIFSuite: tools for HDL code conversion and manipulation," *EURASIP Journal on Embedded Systems*, pp. 1–20, 2010.
34. G. G. Gielen and R. A. Rutenbar, "Computer-aided design of analog and mixed-signal integrated circuits," *Proc. of the IEEE*, vol. 88, no. 12, pp. 1825–1854, 2000.
35. G. Akhras, "Smart materials and smart systems for the future," *Canadian Military Journal*, vol. 1, no. 3, pp. 25–31, 2000.
36. M. Gad-el Hak, *The MEMS handbook*. CRC press, 2001.
37. M. Lora, S. Vinco, and F. Fummi, "A unifying flow to ease smart systems integration," in *High Level Design Validation and Test Workshop (HLDVT), 2016 IEEE International*. IEEE, 2016, pp. 113–120.
38. S. Vinco, V. Guarnieri, and F. Fummi, "Code Manipulation for Virtual Platform Integration," *IEEE Transactions on Computers*, 2015.
39. M. Barnasconi *et al.*, "SystemC AMS extensions user's guide," *Accellera Systems Initiative*, vol. 8, pp. 14–72, 2010.
40. I. Blanco *et al.*, "Smart system case studies," in *Smart Systems Integration and Simulation*. Springer, 2016, pp. 195–227.
41. C. B. Aoun *et al.*, "Pre-simulation elaboration of heterogeneous systems: The SystemC multi-disciplinary virtual prototyping approach," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, jul 2015, pp. 278–285. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7363686
42. K. Kundert, *The designer's guide to Verilog-AMS*. Springer Science & Business Media, 2004.
43. N. J. Godambe and C. J. Shi, "Behavioral Level Noise Modeling and Jitter Simulation of Phase-Locked Loops with Faults Using VHDL-AMS," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 13, no. 1, pp. 7–17, 1998.
44. F. M and B. Z, "Fault Diagnosis in Analog Circuits via Symbolic Analysis Techniques," in *Analog Circuits*. InTech, jan 2013, pp. 237–261.
45. M. Shokouhifar and A. Jalali, "An evolutionary-based methodology for symbolic simplification of analog circuits using genetic algorithm and simulated annealing," *Expert Systems with Applications*, vol. 42, no. 3, pp. 1189–1201, feb 2015.
46. M. L. Crow and M. D. Ilić, "The Waveform Relaxation method for systems of differential/algebraic equations," *Mathematical and Computer Modelling*, vol. 19, no. 12, pp. 67–84, jun 1994.
47. P. Saviz and O. Wing, "Circuit Simulation by Hierarchical Waveform Relaxation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 12, no. 6, pp. 845–860, jun 1993.
48. W. Beyene, "Applications of Multilinear and Waveform Relaxation Methods for Efficient Simulation of Interconnect-Dominated Nonlinear Networks," *IEEE Transactions on Advanced Packaging*, vol. 31, no. 3, pp. 637–648, aug 2008.

49. R. Gillon and N. Bombieri, "D3.2.2 Final Multi-Level Models for Digital Components and Subsystems Public Summary," Smart Components & Smart Systems Integration Consortium, Tech. Rep., 2015.
50. M. J. Schubert, "An analog-node model for VHDL-Based simulation of RF integrated circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 12, pp. 2717–2727, dec 2009.
51. R. Drath and A. Horch, "Industrie 4.0: Hit or hype? [industry forum]," *IEEE Industrial Electronics Magazine*, vol. 8, no. 2, pp. 56–58, jun 2014.
52. S. Centomo, M. Panato, and F. Fummi, "Cyber-physical systems integration in a production line simulator," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, oct 2018.
53. J. Vachalek, L. Bartalsky, O. Rovny, D. Sismisova, M. Morhac, and M. Loksik, "The digital twin of an industrial production line within the industry 4.0 concept," in *2017 21st International Conference on Process Control (PC)*. IEEE, jun 2017.
54. D. Mourtzis, M. Doukas, and D. Bernidaki, "Simulation in manufacturing: Review and challenges," *Procedia CIRP*, vol. 25, pp. 213–229, 2014.
55. "Simulation software survey," 2017. [Online]. Available: https://www.informs.org/ORMS-Today/OR-MS-Today-Software-Surveys/Simulation-Software-Survey
56. F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. of the Workshop on Mobile Cloud Computing*, 2012, pp. 13–16.
57. T. Savolainen, J. Soininen, and B. Silverajan, "IPv6 addressing strategies for IoT," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3511–3519, 2013.
58. M. Li, Z. Yang, and Y. Liu, "Sea depth measurement with restricted floating sensors," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 1, pp. 1–21, aug 2013.
59. N. Bombieri, F. Fummi, and D. Quaglia, "System/network design-space exploration based on TLM for networked embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 9, no. 4, pp. 1–32, mar 2010.
60. P. Sayyah, M. T. Lazarescu, S. Bocchio, E. Ebeid, G. Palermo, D. Quaglia, A. Rosti, and L. Lavagno, "Virtual platform-based design space exploration of power-efficient distributed embedded applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, pp. 49:1–49:25, Apr. 2015.
61. K. Tsilipanos, I. Neokosmidis, and D. Varoutas, "A system of systems framework for the reliability assessment of telecommunications networks," *IEEE Systems Journal*, vol. 7, no. 1, pp. 114–124, 2013.
62. F. Fummi, G. Lovato, D. Quaglia, and F. Stefanni, "Modeling of communication infrastructure for design-space exploration," in *Proc. of Forum on Specification & Design Languages*, Sep. 2010, pp. 1–6.
63. E. Ebeid, F. Fummi, and D. Quaglia, "Model-driven design of network aspects of distributed embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 34, no. 4, pp. 603–614, Apr. 2015.
64. Object Management Group, "MARTE," . [Online]. Available: http://www.omgmarte.org
65. ——, "Unified Modeling Language," . [Online]. Available: http://www.uml.org
66. ——, "SysML," . [Online]. Available: http://www.sysml.org
67. The MathWorks, Inc., "Simulink," . [Online]. Available: http://www.mathworks.com/products/simulink/
68. ——, "Stateflow," . [Online]. Available: http://www.mathworks.com/products/stateflow/

69. Center for Hybrid and Embedded Software System, “Ptolemy,” . [Online]. Available: http://ptolemy.eecs.berkeley.edu/index.htm
70. G. Kahn, “The semantics of a simple language for parallel programming,” *Information Processing*, pp. 471–475, 1974.
71. “IEEE standard for standard SystemC language reference manual,” *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.
72. Transaction Level Modeling Working Group, “OSCI TLM 2.0,” . [Online]. Available: http://www.systemc.org
73. Center for Embedded and Computer Systems, “SpecC,” . [Online]. Available: http://cecs.uci.edu/~{}specc/
74. A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu, “metroii: A design environment for cyber-physical systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, pp. 49:1–49:31, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2435227.2435245
75. A. Bakshi and V. Prasanna, “Algorithm design and synthesis for wireless sensor networks,” in *Proc. of Int. Conf. on Parallel Processing*, 2004, pp. 423–430 vol.1.
76. A. Bonivento, L. P. Carloni, and A. Sangiovanni-Vincentelli, “Platform-based design of wireless sensor networks for industrial applications,” in *Proc. of the Design Automation & Test in Europe Conference*, 2006, pp. 1103–1107.
77. L. Mottola, A. Pathak, A. Bakshi, V. K. Prasanna, and G. P. Picco, “Enabling scope-based interactions in sensor network macroprogramming,” in *Proc. of IEEE Int. Conf. on Mobile Adhoc and Sensor Systems*, 2007, pp. 1–9.
78. A. Puggelli, M. M. R. Mozumdar, L. Lavagno, and A. L. Sangiovanni-Vincentelli, “Routing-aware design of indoor wireless sensor networks using an interactive tool,” *IEEE Systems Journal*, vol. 9, no. 3, pp. 717–727, Sep. 2015.
79. A. Pinto, M. D’Angelo, C. Fischione, E. Scholte, and A. Sangiovanni-Vincentelli, “Synthesis of embedded networks for building automation and control,” in *Proc. of the American Control Conference*, Jun. 2008, pp. 920–925.
80. G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet, “Communication synthesis and HW/SW integration for embedded system design,” in *Proc. of the 6th Int. Workshop on Hardware/Software Codesign*, 1998, pp. 49–53.
81. L. Benini and G. De Micheli, *Networks on Chips: Technology and Tools*. Elsevier, 2006.
82. E. Zahavi, I. Cidon, and A. Kolodny, “Gana: A novel low-cost conflict-free NoC architecture,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 4, pp. 109:1–109:20, Jun. 2013.
83. C. Seiculescu, D. Rahmati, S. Murali, H. Sarbazi-Azad, L. Benini, and G. De Micheli, “Designing best effort networks-on-chip to meet hard latency constraints,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 4, p. 1, Jun. 2013.
84. A. Agarwal, B. Raton, C. Iskander, H.-t. Multisystems, and R. Shankar, “Survey of network on chip (NoC) architectures & contributions,” in *Networks*, vol. 3, no. 1, 2009.
85. U. Y. Ogras and R. Marculescu, “Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach,” in *Proc. of the Design Automation & Test in Europe Conference*, 2005, pp. 352–357.
86. C. E. Rhee, H. Y. Jeong, and S. Ha, “Many-to-many core-switch mapping in 2-D mesh NoC architectures,” in *Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, 2004, pp. 438–443.
87. C. W. Lin, L. Rao, P. Giusto, J. D’Ambrosio, and A. L. Sangiovanni-Vincentelli, “Efficient wire routing and wire sizing for weight minimization of automotive systems,” *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 34, no. 11, pp. 1730–1741, Nov. 2015.

88. S. Xu, R. Kumar, and A. Pinto, “Correct-by-construction and optimal synthesis of beacon-enabled ZigBee network,” *IEEE Trans. on Automation Science and Engineering*, vol. 10, no. 1, pp. 137–144, Jan. 2013.
89. Y.-X. Zhang, K. Takahashi, N. Shiratori, and S. Noguchi, “An interactive protocol synthesis algorithm using a global state transition graph,” *IEEE Transactions on Software Engineering (TSE)*, vol. 14, no. 3, pp. 394–404, Mar. 1988.
90. A. Khoumsi, R. Dssouli, and G. V. Bochmann, “Protocol synthesis for real-time applications,” in *Proc. of Joint Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, 1999, pp. 417–433.
91. H. Yamaguchi, K. Okano, T. Higashino, and K. Taniguchi, “Protocol synthesis from time Petri net based service specifications,” in *Proc. Int. Conf. on Parallel and Distributed Systems*, Dec. 1997, pp. 236–243.
92. R. L. Probert and K. Saleh, “Synthesis of communication protocols: Survey and assessment,” *IEEE Transactions on Computers (TC)*, vol. 40, no. 4, pp. 468–476, Apr. 1991.
93. P. V. Eijk and J. Schot, “An exercise in protocol synthesis,” in *Formal Description Techniques IV. North-Holland*, 1991, pp. 117–131.
94. T. A. Gonsalves and F. A. Tobagi, “Comparative performance of voice/data local area networks,” *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 5, pp. 657–669, Jun. 1989.
95. A. Luntovskyy and A. Schill, “Functionality of wireless network design tools,” in *Proc. of 19th Int. Crimean Conference Microwave Telecommunication Technology*, Sep. 2009, pp. 343–345.
96. Z. Kaleem, T. M. Yoon, and C. Lee, “Energy efficient outdoor light monitoring and control architecture using embedded system,” *IEEE Embedded Systems Letters*, vol. 8, no. 1, pp. 18–21, Mar. 2016.
97. K. Das, P. Zand, and P. Havinga, “Industrial wireless monitoring with energy-harvesting devices,” *IEEE Internet Computing*, vol. 21, no. 1, pp. 12–20, Jan 2017.
98. Y. Y. Chen, C. H. Hsu, and K. L. Leu, “SoC-level risk assessment using FMEA approach in system design with systemC,” in *Proc. of IEEE SIES 2009*, 2009, pp. 82–89.
99. A. Sherer, J. Rose, and R. Oddone, “Ensuring functional safety compliance for ISO 26262,” in *Proc. of IEEE/ACM DAC 2015*, 2015, pp. 1–3.
100. R. Weissnegger, C. Kreiner, M. Pistauer, K. Römer, and C. Steger, “Sharc - simulation and verification of hierarchical embedded microelectronic systems,” *Procedia Computer Science*, vol. 109, pp. 392–399, 2017, 8th International Conference on Ambient Systems, Networks and Technologies, ANT-2017 and the 7th International Conference on Sustainable Energy Information Technology, SEIT 2017, 16-19 May 2017, Madeira, Portugal.
101. A. Fin and F. Fummi, “A VHDL error simulator for functional test generation,” in *Proc. of the IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2000, pp. 390–395.
102. R. Mariani, G. Boschi, and F. Colucci, “Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508,” in *Proc. of the IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, apr 2007, pp. 492–497.
103. M. Soma, “Challenges in analog and mixed-signal fault models,” *IEEE Circuits and Devices Magazine*, vol. 12, no. 1, pp. 16–19, 1996.

104. N. Bombieri, F. Fummi, and V. Guarnieri, “FAST: An RTL fault simulation framework based on RTL-To-TLM abstraction,” *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 28, no. 4, pp. 495–510, 2012.
105. B. Stroustrup, *The C++ Programming*, 1986.
106. H. T. Vierhaus, W. Meyer, and U. Glaser, “CMOS bridges and resistive transistor faults: IDDQ versus delay effects,” in *Proc. of IEEE International Test Conference - (ITC)*, Oct 1993, pp. 83–91.
107. J. M. Acken, “Special Applications of the Voting Model for Bridging Faults,” *IEEE Journal of Solid-State Circuits*, vol. 29, no. 3, pp. 263–270, 1994.
108. Y. J. Chang, C. L. Lee, J. E. Chen, and C. Su, “Behavior-level fault model for the closed-loop operational amplifier,” *Journal of Information Science and Engineering*, vol. 16, no. 5, pp. 751–766, 2000.
109. C. Henderson, J. Soden, and C. Hawkins, “The Behavior and Testing Implications of Cmos Ic Logic Gate Open Circuits,” *Proc. International Test Conference*, no. 1, pp. 302–310, 1991.
110. R. Leveugle and A. Ammari, “Early SEU Fault Injection in Digital, Analog and Mixed Signal Circuits: A Global Flow,” in *Proc. of the IEEE/ACM Design, Automation & Test in Europe Conference & Exhibition (DATE)*, vol. 1. IEEE Comput. Soc, 2004, pp. 590–595.
111. S. N. Ahmadian and S. G. Miremadi, “Fault injection in mixed-signal environment using behavioral fault modeling in Verilog-A,” in *Proc. of the IEEE International Workshop on Behavioral Modeling and Simulation, BMAS*. IEEE, sep 2010, pp. 69–74. [Online]. Available: http://ieeexplore.ieee.org/document/6156601/
112. M. J. Barragan, H. G. Stratigopoulos, S. Mir, H. Le-Gall, N. Bhargava, and A. Bal, “Practical simulation flow for evaluating analog/mixed-signal test techniques,” *IEEE Design & Test*, vol. 33, no. 6, pp. 46–54, Dec 2016.
113. A. Singhee and R. A. Rutenbar, “Statistical blockade: Very fast statistical simulation and modeling of rare circuit events and its application to memory design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 8, pp. 1176–1189, 2009.
114. D. Quaglia and F. Stefanni, “SystemC Network Simulation Library – version 2,” 2013, URL: http://sourceforge.net/projects/scnsl.
115. W. Du, F. Mieyeville, and D. Navarro, “Idea1: A systemc-based system-level simulator for wireless sensor networks,” in *2010 IEEE International Conference on Wireless Communications, Networking and Information Security*, June 2010, pp. 618–622.
116. International Electrotechnical Commission, “IEC 61508 : Functional safety of electrical/electronic/ programmable electronic safety-related systems,” pp. 7–13, 2010.

A

Verilog-AMS designs

Listing A.1: Analog to Digital Converter (ADC) written in Verilog-AMS.

```
module adc(out, in, gnd, clk);
    // PARAMTERS -----------------------
    // Resolution (bits)
    parameter integer bits   = 8 from [1:24];
    // Minimum input voltage (V)
    parameter real vmin      = 0.0;
    // Maximum input voltage (V)
    parameter real vmax      = 3.3 from (vmin:inf);
    // PORTS ---------------------------
    output [bits-1:0] out;
    input in, gnd, clk;
    // NODES ---------------------------
    reg [bits-1:0] out;
    electrical in, gnd;
    // LOCAL ---------------------------
    localparam midpoint = ((vmax-vmin) / 2);
    reg over;
    real sample;
    integer i;
    // BEHAVIOR ------------------------
    always @(posedge clk) begin
        sample = V(in, gnd);
        for (i=bits-1; i>= 0; i = i - 1) begin
            over = (sample >= midpoint);
            if (over) begin
                sample = sample - midpoint;
            end
            sample = 2.0*sample;
            out[i] <= over;
        end
    end
endmodule
```

Listing A.2: Ideal operational amplifier written in Verilog-AMS.

```
`define dB2dec(x) pow(10, x/20)
module opamp(out, in, gnd);
    // PARAMTERS -----------------------
    parameter real gain           = 10  from [0:inf);
    parameter real three_db_freq = 50k from (0:inf);
    parameter real rin            = 1M  from (0:inf);
    parameter real cin            = 1n  from [0:inf);
    parameter real iout_max       = 10u from (0:inf);
    parameter real rout           = 100 from (0:inf);
    parameter real volc           = 5   from (0:inf);
    parameter real vdd            = 3.3;
    parameter real rgnd           = 1e09 from (0:inf);
    // PORTS ---------------------------
    inout out, in, gnd;
    // NODES ---------------------------
    electrical out, in, gnd, mid, B;
    // LOCAL ---------------------------
    real vb, iout, A;
    // BEHAVIOR ------------------------
    analog begin
        // Input Stage
        I(in, mid) <+ V(in, mid)  / rin;
        I(in, mid) <+ ddt(V(in, mid)) * cin;
        I(mid, gnd) <+ V(mid, gnd)  / rgnd;
        // Dominant Pole
        A = laplace_nd(V(in) * `dB2dec(gain),
                       {1},
                       {1, 1 / (three_db_freq*`M_TWO_PI)});
        if      (A > +vdd) A = +vdd;
        else if (A < -vdd) A = -vdd;
        V(B, gnd) <+ A;
        // Output current evaluation and clipping
        I(B, out) <+ V(B, out) / rout;
        iout = V(B, out) / rout;
        if (iout > +iout_max) iout = -iout_max;
        else if (iout < -iout_max) iout = +iout_max;
        else iout = -iout;
        I(B, out) <+ iout;
    end
endmodule
```

Listing A.3: Motor description written in Verilog-AMS.

```
module motor(shaft_position, absorb, p, n);
    // PARAMTERS -----------------------
    parameter real km = 4.5;    // motor constant (V-s/rad)
    parameter real kf = 6.2;    // flux constant (N-m/A)
    parameter real j = 0.004;   // inertia of shaft (N-m-s2/rad)
    parameter real d = 0.1;     // drag (friction) (N-m-s/rad)
    parameter real r = 5.0;     // motor winding resistance (Ohms)
    parameter real l = 0.02;    // motor winding inductance (H)
    // PORTS --------------------------
    output shaft_position, absorb;
    input p, n;
    // NODES --------------------------
    rotational shaft_position;
    electrical absorb, p, n;
    // Internal nodes.
    electrical n1, n2;
    rotational_omega shaft, rgnd;
    // Reference nodes.
    ground rgnd;
    // BRANCHES ------------------------
    branch (p,  n1) Vm;
    branch (n1, n2) R1;
    branch (n2, n)  L1;
    branch (shaft, rgnd) bshaft;
    branch (shaft_position, rgnd) bshaftp;
    // BEHAVIOR ------------------------
    analog begin
        // Electrical model of the motor winding.
        V(Vm) <+ km * Omega(bshaft);
        V(R1) <+  r * I(R1);
        V(L1) <+  l * ddt(I(L1));
        // Physical model of the shaft (keep like this).
        Tau(bshaft) <+ + kf * I(Vm);
        Tau(bshaft) <+ - d  * Omega(bshaft) - j * ddt(Omega(bshaft));
        // Equation for conversion to degrees.
        Theta(bshaftp) <+ (180 * idt(Omega(bshaft), 0)) / `M_PI;
        //     deg : rad = 180 : 3.14
        //     deg = 180 * rad / 3.14
        // Provide a measure of the motor response.
        V(absorb) <+ V(Vm);
    end
endmodule
```

Listing A.4: Digital to Analog Converter (DAC) written in Verilog-AMS.

```
module dac(out, gnd, in, clk);
    // PARAMTERS -----------------------
    /// Resolution. (bits)
    parameter integer bits = 8 from [1:24];
    /// Minimum input voltage. (V)
    parameter real vmin = 0.0;
    /// Maximum input voltage. (V)
    parameter real vmax = 3.3 from (vmin:inf);
    /// Delay from clock edge to output. (s)
    parameter real td = 0;
    /// Transition time of output. (s)
    parameter real tt = 0;
    // PORTS ---------------------------
    output out;
    input gnd, [bits-1:0] in, clk;
    // NODES ---------------------------
    voltage out;
    electrical gnd;
    logic [bits-1:0] in;
    logic clk;
    // LOCAL ---------------------------
    localparam real fullscale = vmax - vmin;
    real aout;
    integer weight, i;
    // BEHAVIOR ------------------------
    always @(posedge clk) begin
        aout = 0.0;
        weight = 2;
        for (i = bits - 1; i >= 0; i = i - 1) begin
            if (in[i]) begin
                aout = aout + (fullscale / weight);
            end
            weight = weight * 2;
        end
    end
    /// Smoothly moves the output between the values.
    analog V(out, gnd) <+ transition(aout + vmin, td, tt);
endmodule
```

Listing A.5: TransImpedence Amplifier (TIA) written in Verilog-AMS.

```
module tia(out, in, gnd);
    // PARAMTERS -----------------------
    parameter real rin = 1e4;
    parameter real gain = 10;
    parameter real f_cut = 1e3;
    parameter real threshold = 3.3;
    // PORTS ---------------------------
    inout in, out, gnd;
    // NODES ---------------------------
    electrical in, out, gnd;
    // LOCAL ---------------------------
    real vint;
    // BEHAVIOR ------------------------
    analog begin
        I(in, gnd) <+ V(in, gnd) / rin;
        vint = 1.5 + gain * V(in, gnd);
        if ( vint > +threshold ) begin
            vint = +threshold;
        end
        if ( vint < -threshold ) begin
            vint = -threshold;
        end
        V(out, gnd) <+ laplace_nd(V(int),
            {1}, {1 ,1/(`M_TWO_PI * f_cut)});
    end
endmodule
```

Listing A.6: Comparator written in Verilog-AMS

```
module comparator(alarm_sig, in, pref, nref);
    // PARAMTERS -----------------------
    /// Delay from clock edge to output (s).
    parameter real td = 0 from [0:inf);
    /// Transition time of output (s).
    parameter real tt = 0 from [0:inf);
    // PORTS ---------------------------
    inout alarm_sig, in, pref, nref;
    // NODES ---------------------------
    voltage in, pref, nref;
    reg alarm_sig = 0;
    // BEHAVIOR ------------------------
    analog begin
        @(cross(V(in) - V(pref), +1)) begin
            alarm_sig = 1;
        end
        @(cross(V(in) - V(nref), -1)) begin
            alarm_sig = 1;
        end
    end
    /// If the output alarm signal is high, lower it after 20 nanoseconds.
    always @(alarm_sig) begin
        if (alarm_sig) begin
            alarm_sig = #(20) 0;
        end
    end
endmodule
```

Listing A.7: Behavioral-level CMOS inverter written in Verilog-AMS.

```
`include "constants.vams"
`include "disciplines.vams"
module inverter(q, a, vdd, vss, sub);
    // Input, output ports and supplies.
    electrical output q;
    electrical input  a, vdd, vss, sub;
    // Voltage thresholds.
    parameter real vth_u = 1.65 from [0:inf);
    parameter real vth_d = 1.65 from [0:inf);
    // Smoothing factors.
    parameter real smth_u = 0.1 from [0:inf);
    parameter real smth_d = 0.1 from [0:inf);
    // Inverter parameters.
    parameter real C_a_vdd_h = 1.0 from [0:inf);
    parameter real C_a_vdd_l = 1.0 from [0:inf);
    parameter real C_a_vss_h = 1.0 from [0:inf);
    parameter real C_a_vss_l = 1.0 from [0:inf);
    parameter real C_vdd_q_h = 1.0 from [0:inf);
    parameter real C_vdd_q_l = 1.0 from [0:inf);
    parameter real C_vss_q_h = 1.0 from [0:inf);
    parameter real C_vss_q_l = 1.0 from [0:inf);
    parameter real R_vdd_q_h = 1.0 from [0:inf);
    parameter real R_vdd_q_l = 1.0 from [0:inf);
    parameter real R_vss_q_h = 1.0 from [0:inf);
    parameter real R_vss_q_l = 1.0 from [0:inf);
    /// @brief Smooth switching function with tanh.
    analog function real tanhsw;
        input x, swpt, smth;
        real x, swpt, smth;
        begin
            tanhsw = 0.5 + 0.5 * tanh((x-swpt)/smth);
        end
    endfunction
    // Variable capacitances and resistances.
    real C_a_vdd, C_a_vss;
    real C_vdd_q, C_vss_q;
    real R_vdd_q, R_vss_q;
    // Support variables used to perform the switching.
    real x, x_u, x_d;
    analog begin
        // Switching point evaluation.
        x = V(a);
        x_u = tanhsw(x, vth_u, smth_u);
        x_d = tanhsw(x, vth_d, smth_d);
        // Perform smooth switching.
        C_a_vdd = (x_u*C_a_vdd_h) + ((1-x_u)*C_a_vdd_l);
        C_a_vss = (x_d*C_a_vss_h) + ((1-x_d)*C_a_vss_l);
        R_vdd_q = (x_u*R_vdd_q_h) + ((1-x_u)*R_vdd_q_l);
        C_vdd_q = (x_u*C_vdd_q_h) + ((1-x_u)*C_vdd_q_l);
        R_vss_q = (x_d*R_vss_q_h) + ((1-x_d)*R_vss_q_l);
        C_vss_q = (x_d*C_vss_q_h) + ((1-x_d)*C_vss_q_l);
        // Components instantiation.
        I(a, vdd) <+ ddt(V(a, vdd)) * C_a_vdd;
        I(a, vss) <+ ddt(V(a, vss)) * C_a_vss;
        I(vdd, q) <+ V(vdd, q) / R_vdd_q;
        I(vss, q) <+ V(vss, q) / R_vss_q;
        I(vdd, q) <+ ddt(V(vdd, q)) * C_vdd_q;
        I(vss, q) <+ ddt(V(vss, q)) * C_vss_q;
    end
endmodule
```

B

C++ algorithms

Listing B.1: Functions used to deal with equality between values of type *double*. Beside the two variables to compare, the function receives the value of *tolerance*, used to tune the comparison's accuracy.

```cpp
template<typename T1, typename T2>
inline bool dbl_equal(T1 const & a, T2 const & b, double tolerance = 1E-09)
{
    return (static_cast<int>(fmax(fabs(a), fabs(b)) / tolerance))
            ? (fabs(a - b) < tolerance) : true;
}

template<typename T1, typename T2>
inline bool dbl_lequal(T1 const & a, T2 const & b, double tolerance = 1E-09)
{
    return dbl_equal(a, b, tolerance) || (a < b);
}

template<typename T1, typename T2>
inline bool dbl_gequal(T1 const & a, T2 const & b, double tolerance =1E-09)
{
    return dbl_equal(a, b, tolerance) || (a > b);
}
```

Listing B.2: Functions used to generate random real values.

```
/// @brief Generates a random real value between min and max.
/// @tparam DataType The type of the boundaries.
/// @param min Lower bound.
/// @param max Upper bound.
/// @return The random value between lower and upper bounds.
template<typename DataType>
inline DataType RealRand(DataType min, DataType max)
{
    assert(min <= max && "Trying to randomize in non-valid range.");
    using UniformDist = std::uniform_real_distribution<>;
    thread_local static std::mt19937 mt(std::random_device{}());
    thread_local static UniformDist urd;
    return urd(mt, typename UniformDist::param_type(min, max));
}
```