IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 20 August 2023, accepted 12 October 2023, date of publication 17 October 2023, date of current version 25 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3325349

THEORY

A Model of Design for Computing Systems: A Categorical Approach

TAGE MOHAMMADAT, (Senior Member, IEEE)
Embedded Computing Systems and KTH Royal Institute of Technology, 114 28 Stockholm, Sweden
e-mail: tage@embedded-computing.eu

ABSTRACT This paper introduces the model of design (MoD), a framework that leverages category theory to study the design and development of computer-driven systems, to the academic and engineering communities dealing with computer systems. The model of design aims to offer a minimal framework for modelling the design and development of embedded computation across domains and abstractions, focusing on functional and extra-functional aspects as well as overarching concerns for automaticity, correctness and reuse. This nuanced approach provides insights into the theory and practice of computer systems design.

INDEX TERMS Computing systems, computer-aided design (CAD), electronic design automation (EDA), embedded system design, architectural design, model-driven engineering, domain-specific modelling languages, model of computation, system-level design, hardware/software co-design.

I. INTRODUCTION AND BACKGROUND

Since the advent of the digital revolution in the twentieth century, computing has become ubiquitous in nearly every facet of human society, permeating our daily lives, urban infrastructure, global connectivity, and extraterrestrial endeavours. Driven by the market and benefiting from economies of scale, the sheer quantity of computers – whether in familiar forms or embedded within other products – has surpassed the global human population by two to three times, with a growth rate outpacing that of the human population, as indicated by the Cisco internet report [1]. Stanley Mazor, one of the co-inventors of the first microcomputer system [2], eloquently expressed this sentiment in his memoir published in the IEEE Solid-State Circuit Magazine in 2009: ''In 1960 - ten years before Intel developed the first single-chip CPU (microcomputer central processing unit) - the revolution that would ensue was inconceivable: the cost of computing dropped by a factor of a million, modes of personal communication changed forever, and intelligent machines took over processes in manufacturing, transportation, medicine - virtually every aspect of our lives.'' [3].

The rapid proliferation of computers and the growing demand for computing resources did not only, as Mazor remarked, elude the pioneers, but have also presented challenges to our capacity to engineer the necessary machinery for their production. To overcome these challenges, we employ computers to assist us in the development and manufacturing of next-generation computers and computer applications. Computer-assisted development tools and electronic design automation help tackle the engineering challenge by leveraging domain-specific computer programs. By utilising high-level design and programming languages, we are able to write precise design descriptions and algorithms for computer programs that are intelligible to engineers. These descriptions can then be iteratively refined, optimised, analysed, and synthesised (by computer programs) into machine-level representations that are understood by computers. The computers, embedded within fabrication foundries, assembly lines, vehicles, data centres and personal devices, then process these machine codes to produce useful apparatus and executable applications at large scales. This engineering ecosystem is made possible by the continuous and compound advancements in information theories, computer languages, communication technologies, device modelling and design automation strategies, simulation and compilation technologies, verification and optimisation methods, and computerised instrumentation & control systems, which evolve in tandem to support and enhance the development process of computer systems and applications.

The associate editor coordinating the review of this manuscript and approving it for publication was Ludovico Minati.

To cope with the increasing complexity of computer system design engineering ecosystem and the expanding application domains, design processes have been successively adjusted by raising the level of abstraction for engineers, segregating design concerns, and breaking down the scope into more manageable subsets across teams. Researchers and practitioners such as Lee and Sangiovanni-Vincentelli [4], [5], [6], [7], [8], [9], have articulated how computers and applications are engineered using modelling languages at higher levels of abstraction incorporating frameworks such as model-driven development, component-based integration, and platform-based design. In each engineering domain, domain-specific modelling languages are utilised to refine and integrate technical specifications, addressing particular design considerations to achieve the desired design outcome at a particular level of abstraction. In navigating through the array of differing modelling languages, the application of metamodelling techniques and metaprogrammable tools becomes inevitable, and the introduction of more domain-specific modelling languages is necessitated. However, this diversification leads to a complex spectrum of design paradigms, which can sometimes present conflicting methodologies. They conclude that the key to managing design complexity lies in employing structured and formal design methodologies that can harmoniously integrate the various facets of the multi-scale design space, be it behavioural, spatial, or temporal. These methodologies must provide suitable abstractions to tackle the inherent complexity and ensure that the resultant implementations are correct-by-construction from the outset.

Recognising the need for further development and, in particular, for unifying methods in designing computer-driven systems, computer scientist Joseph Sifakis, known for his works in model checking, has advocated for theories of design in his work on system design automation [10] for embedded and cyber-physical systems. Like other researchers, he envisions that designing embedded and cyber-physical systems should be a process that is correct-by-construction. However, he acknowledges that such formalisation poses significant theoretical challenges, including the conceptualisation of needs, formal requirement expression, and the development of functionally correct and optimised implementations on specific platforms. Despite the immense potential, this venture has received, according to Joseph Sifakis, limited attention from scientific communities, partly due to the academic world's preference for simple and elegant theories and the multidisciplinary nature of the field. Sifakis concludes that achieving such formalisation requires consistent integration of heterogeneous system models that support different levels of abstraction, which encompass logic, algorithms, programs, and physical system models.

In agreement with the aforementioned views, we recognise that as the evolution of engineering paradigms continues and the intersection of disciplines increases, the task of developing a comprehensive design framework for computing systems, applicable across various domains and stages of development, becomes ever more arduous. A meticulously crafted framework that strikes a balance between abstraction and precision can equip system designers, architects, business owners, and engineers with the means to aptly comprehend the intricacies of diverse computer products and applications. Such a framework allows them to focus selectively on particular areas of interest while preserving a generalised understanding within the broader context. This becomes especially crucial in the emergent landscape where computer programs are integrated with large language models equipped with natural language processing capabilities. Engineers can then prioritise comprehending overarching aspects of design problems and the development of computer applications or systems from the perspective of specifications, rather than needing to primary the minutiae of machine implementations, which can be automatically generated by computers.

In this article, we introduce the *model of design*, a framework established upon the foundations of category theory [11]. The model of design attempts to scalably assist in the comprehension and navigation of the intricacies inherent in computer engineering problems. As an abstract mathematical branch, category theory presents robust toolsets for rationalising about abstract objects via their interrelationships. By applying category theory's lens to the complex landscape of computer system design models and methodologies, we can distill properties useful for system design, thereby enabling reasoning around high-level notions such as compositionality, equivalence, and coherence. Leveraging the model-of-design concepts, we posit that vital components for correct-by-construction design flows and automation of computer systems and applications can be discerned. Although the model-of-design framework is not intended as an exhaustive, end-to-end correct-by-construction solution for all computing-related engineering disciplines, it seeks to provide a language encapsulating existing design paradigms and coalescing diverse domain perspectives within a rigorously delineated framework. Owing to its abstract nature, we believe the model of design can illuminate fresh perspectives and provide opportunities for future exploration and innovation beyond traditional computing systems.

The remainder of this manuscript is structured as follows: Section II introduces the necessary preliminaries to prepare the reader for the formal definitions outlined in the subsequent sections. In Section III, we lay the foundation for our proposed framework. Subsection III-A delves into the essential elements of this framework, encompassing specifications, architecture, implementation, evaluation, and design decisions, and concludes by providing a comprehensive definition of our model of design. Subsections III-B and III-C detail the emergent properties of our model and the corollaries derived from these properties, respectively. Afterwards, Section IV situates our work within the context of existing literature, offering a detailed discussion. The

paper concludes with Section V, where we synthesise the key insights from our model of design, discuss its potential applications, and identify promising avenues for future research.

II. PRELIMINARIES

To establish our conceptual framework, we begin by revisiting three foundational constructs: modelling languages, abstraction spaces, and categorical axioms as follows:

Foundational Construct 1. Language (Modelling Language)

A *modelling language*, or simply a language L is a tuple, $L = \langle \Sigma, G, S \rangle$ where:

- Σ is a finite set of strings formed over an alphabet and adheres to the rules defined by G. Each string in Σ represents a well-formed sentence of the language.
- G is a grammar defined as $G = \langle \alpha, V, P, S \rangle$, where:
  - α is a finite set of terminals or symbols.
  - V is a finite set of variables distinct from α.
  - P is a set of production rules, with each rule mapping a variable to a string of symbols and variables.
  - $S \in V$ is the start variable.
- S is a mapping from Σ to the semantics or meaning associated with each string in Σ.

The alphabet α is represented as $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_k\}$ with cardinality $|\alpha| = k$. A string in Σ is a sequence, potentially repetitive, of symbols from α that has semantic meaning as defined by S.

A *file* is any binary or alternative representation, like ASCII or UTF-8, that adheres to the grammar G of the language. Files may extend the original language for storage metadata, such as EOF markers. A *library* is an organised collection of one or more such files.

Remark 1 (Regarding Languages).

- Noam Chomsky categorised languages into several types: 1) regular expressions handled by finite state machines, 2) context-free and context-sensitive grammars that can be addressed by stack-based computing machines (push-down automata) or by linear bounded automata, and 3) enumerable languages, which can be managed by Turing machines and random access memory (RAM) based computers [12], [13]. The reasoning behind employing languages as a fundamental concept in computer design lies in their ability to capture varying complexity levels, which necessitate different complexity levels of computers for their analysis, compilation, and synthesis into implementations across a range of abstraction and development stages.
- This formal definition of language serves several purposes: 1) to distinguish our usage from the informal ones, such as those seen in natural languages used in requirement engineering for computing system designs, 2) to enable the manipulation or translation of the language through well-established theories in formal languages and semantics, and 3) to adopt a broad approach, treating formats, computer scripts, programs, mathematical formulas, templates, algorithmic representations, data types, graphs, logical/arithmetic operations, and abstract models as languages in a manner that is more formal than non-formal languages.
- The grammar and semantics of a language are typically understood within the context of its application. However, as indicated by Harel and Rumpe [14], the function of semantics in a (modelling) language goes beyond the metamodel of the language, its context conditions, or its execution environment.
- It is important to note that the execution and operational semantics of a language need not be incorporated in the semantics unless the language is a computer programming language. In such cases, execution semantics are crucial for its purpose. This remark aims to separate the definition of the (executable) language from its implementation for correct execution on a computer.
- Fundamentally, languages can be composite, wherein an extended version of a language or a combination of two or more languages still form a language. This is represented as a set of strings of alphabets that follow a grammar-generating meaning.
- Operations on (possibly) composite languages such as product, union, intersections, difference, and cardinality may not embody the usual understanding of the operation in set theory but rather a specialised understanding in the category theory sense [11]. In other words, an operation on a language L is an operation over a subset of the Kleene closure (*) over the alphabet that adheres to composition rules.

Example 1 (Languages). Typically, languages are expressed in documents capturing both the alphabets (lexical, terminals, symbols or vocabulary), grammar (syntax, valid expressions or statements) and semantics together. Examples include the λ calculus (formal system), ISO/IEC8859 8-bit character encodings and ANSI INCITS 4-1986[R2017] 7-Bit ASCII (formats), IEEE 1666 SystemC specifications and ISO/IEC 14882:2020 programming language C++, OMG unified modelling language (UML) (modelling language), the defacto standard graphic design system stream format (GDSII) –database for manufacturing electronic chips), the language reference for simulation programs with integrated circuits emphasis (SPICE) –analysis and design of circuits, IEEE 1076-2019 – very high-speed integrated circuit hardware description language (VHSIC HDL or VHDL – a hardware description language).

Applying principles from category theory, languages can be conceived as categories articulated through ontology logs. Here, the 'objects' are the sentences or expressions within the language, while 'morphisms' are the valid transformations from one sentence to another, adhering to the rules of the language's grammar. This perspective fosters a nuanced understanding of the structural correlations between various sentences in the language and how they may be transformed, providing a solid foundation for formal language analysis.

Ontology logs function as a conduit to express these categories. In the realms of computer and information sciences, ontology encapsulates a conceptualisation specification, deployed to reason about the properties of a domain. It essentially characterises the domain in terms of its objects and their interrelations. Ontology logs furnish a formal and explicit specification of a collective conceptualisation, thereby structuring knowledge about specific domains.

For instance, we could view each sentence in a language as an 'object' in the category, and the potential transformations between these sentences, in accordance with the language's rules, as 'morphisms'. The ontology log methodically captures these relationships, facilitating computational reasoning within the language. Note, further details on category theory, which provide a theoretical backbone for this discussion, will be presented in Foundational Construct 3.

Foundational Construct 2. Abstraction Spaces
Let *abstraction spaces* $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$ be described as a modelling language $L = \langle \Sigma, G, S \rangle$, where:

- Σ encompasses constructs that are consistent across both the main abstraction space and its sub-spaces.
- G is the grammar detailing how these constructs can be combined, applicable to both spaces and their sub-varieties.
- S provides the semantics, detailing the interpretation or meaning for each construct in Σ, suitable for overarching spaces and their finer subdivisions.

This language captures the abstraction spaces, permitting further granularity through sub-spaces, denoted as $\mathbb{S}_{\mathbb{A}}$. Each sub-space inherits the overarching characteristics of its parent space but introduces additional or modified constructs, grammar rules, and semantics, reflecting the depth and breadth of design details specific to its domain.

Importantly, both abstraction spaces and their encapsulated sub-spaces are designed to address arbitrary engineering or practical choices of levels or hierarchies, contingent on the design context, industry domain, and the specific application or system under consideration.

Drawing from [15], [16], we classify the following abstraction spaces and potential subspaces therein:

1) **S/T Space**: symbolises the *system/transaction* space. It covers constructs ranging from transaction-level models (TLM) to systems-of-systems. Within this, potential sub-spaces might delve deeper into specific transaction types or system hierarchies. The criteria distinguishing this space are that information can be represented in transactions or tokens, or spacetime is portrayed using quanta. This space exceeds the register-transfer level, considering abstract pseudocode algorithms on abstract computing machines with perfect synchrony hypothesis or expressing time using abstract time quanta. This space includes what the international semiconductor technology roadmap (ITRS) terms as the electronic system level, which could also house industry-dependent sub-spaces in areas like avionics, communications, and automotive industries.
2) **RT Space**: denotes the *register-transfer* space. It spans between logic layers of abstractions to register transfer levels. Distinguishing features of this space include the representation of information in bits or discrete time units such as cycles, with no explicit spatial component. This space encapsulates instruction set architectures and micro-architectural implementation models, defining data types, operations, and execution models explicitly. Possible sub-spaces in this space within the electronic design automation (EDA) community may include intellectual property (IP) design blocks, gate-level, standard-cells level, and logic level.
3) **C Space**: represents the *circuit* space, detailing constructs from digital switches to analogue circuits, mixed-signal circuits, and beyond to encapsulate other physical computing systems such as chemical, biological, photonic, or mechanical systems. This space is characterised by the capacity to represent information as energy potential (voltage) and electric current, chemical changes, photonic signals, mechanical displacement, or other physical phenomena. In this space, we can denote time continuously, without explicit spatial representation. This abstraction space embraces circuits of discrete components, monolithic circuits, as well as chemical, photonic, or mechanical systems, without distinguishing among them, taking into account the terminal characteristics of circuit element equivalents within the system. Possible sub-spaces include switches, stick diagrams, and other equivalent representations for non-electronic systems.

4) **P Space**: is distinguished by the representation of information as energy or the explicit representation of spacetime. It includes the physical properties of materials, semiconductor physics, and the interaction of energy with spacetime. Different scales of space and time can be represented in this space to capture various details and complexities. Potential sub-spaces could include networked racks in a data centre, interconnected discrete components as in printed circuit boards, and integrated circuits at various scales.

Remark 2 (On the choice of abstraction spaces).

- When a model is said to exist within an abstraction space (or a sub-space), it signifies how computation or computing can be related to the granularity of information representation in relation to spacetime.
- The purpose of this definition is to confine the use of abstraction spaces to the most fundamental categories that uniquely characterise information processing (computing) in relation to spacetime (and possibly energy/matter).
- The most prevalent concept of abstraction spaces over the past four decades, as outlined by [15], [17], includes, generally speaking, system level (or system of systems), processor level, register-transfer level, logical level, transistor/circuit level, and physical implementation. These levels span three domains: behavioural/functional, structural/architectural, and physical/layout/geometric. Although these definitions have been beneficial for the design of very large-scale integrated (VLSI) systems and integrated circuits (IC), and have influenced the development of computer-aided design (CAD) tools and electronic design automation (EDA) engineering, they might not be fundamental nor minimal. Industrial and application domains such as automotive (e.g., AutoSAR) and computer networking (open systems interconnects, OSI) have evolved to establish their own standards for defining domain-specific abstraction spaces. Therefore, since embedded computer systems design hinges on application/industrial domain specificity and electronic design technologies, it seems reasonable to define a fundamental and minimal language for abstraction spaces that aligns with existing definitions.
- The granularity of expressing information with respect to spacetime is the primary criterion considered for the differentiation of abstraction spaces. For instance, in the system/transaction space, information is expressed by transactions and spacetime is expressed using quanta. In the register-transfer space, information is represented as binary digits (bits), while spacetime is denoted as discrete time units, such as clock cycles. In the circuit space, information is represented as voltage/current waveforms and time is continuous. Lastly, in the physical space, information is represented as energy waveforms, and both space and time are explicitly considered. To manage system complexity, we allow abstraction spaces to have sub-spaces that can be used to express hierarchies.
- Although the resulting definition, a fusion of various abstraction concepts, may be overly abstract for immediate application in specific industrial domains, it could be valuable for capturing the generality of the concepts of abstraction and hierarchies for the fundamental analysis of their limitations and trade-offs.
- In theory, the fundamental distinction among the various abstraction spaces might be helpful. However, in practice, system design often spans multiple abstraction spaces, and many design techniques might require information from lower-level spaces or hierarchies to make informed decisions at higher levels. An example of this is the system-circuit cross-space design technique, dynamic voltage scaling (DVS), used at the system level for low-power dynamic computer system design. Embedded and cyber-physical systems are another example where multiple abstractions are exposed.
- Transaction abstraction is considered equivalent to system level and systems of systems level. This is mainly because with the standardisation of SystemC that encapsulates timing aspects in a spectrum of timed to untimed and information in transactions, the distinction of abstraction spaces above the transaction level becomes less fundamental with respect to information and spacetime.
- The consideration of the physical space as an abstraction space, as opposed to a view or a domain as in the Gajski-Kuhn Y-chart, is motivated by the observation that computer systems and networks typically regard the physical details of the computer/network as the most detailed level with respect to the electrical and mechanical characteristics of computing systems constituents. Furthermore, in analogue/radio frequency (RF)/mixed-signal circuits and digital systems, the physical aspects are the closest to the manufactured/implementation.
- It is natural for computer system design to comprise cross-space aspects. The distinction and vocabulary for the abstraction spaces make it possible to outline design activities with respect to the abstraction axis. Obviously, when design complexity extends beyond abstraction space boundaries and becomes 'across-layers' or 'cross-level', the ensuing complexities and concerns are combined.

Remark 3 (Candidate abstraction spaces).

- Y-Chart: 1) Within the behavioural and structural domains, systems, processor components, algorithms and transaction level modelling map to the S/T Space. 2) Within the behavioural and structural domains,

logic functions, boolean logic, arithmetic and logic units (ALU), RT, gates netlists, and flip flops map to the RT Space. 3) Within the behavioural and structural domains, system transfer functions, stick/switch diagrams, standard-cells complementary metal oxide semiconductor (CMOS) based transistor circuits map to the C Space. 4) The physical domain maps to the P Space where the different levels of complexities map to the different sub-spaces within the P Space.

- V-Chart: The V-Chart describes development stages rather than fundamental abstraction spaces: requirement and architectural analysis, system design, unit level coding and implementation, unit integration, validation and testing. We view the level of abstraction within the V-chart as applicable to all of the four spaces described herewith: 1) S/T Space 2) RT Space 3) C Space and 4) P Space.
- Double-roof Model (including a computer architecture taxonomy): 1) Systems, tasks and components map to the S/T Space. 2) Instructions, instruction set architecture (ISA), micro-architecture, gates, logic, RTL map to the RT Space. The double-roof model does not provide explicit details about the C Space or P Space.
- OSI (Networks domain): 1) Applications, presentation and session spaces map to the S/T Space. 2) The transport space, network and data link spaces map to the RT Space. 4) The physical space maps to the C Space and the P Space.
- AUTOSAR Spaces (Automotive domain): 1) Applications and run-time environment (RTE), operating system services map to the S/T Space. 2) Tasks/processes, communication, device drivers, networks drivers, hardware/firmware abstraction spaces, assuming they are at the binary level, map to the RT Space. 3) Vehicle, electronic controller units (ECU) and micro-controllers (MC) map to the C Space and P Space.

Example 2 (Abstraction spaces). An abstraction language over four abstraction spaces (S/T, RT, C, P) could describe the following vocabulary:

- S/T Space: This space contains entities such as SystemC approach to modelling, traditional dataflow model-of-computation theories, UML based models, Simulink based models, and architectural languages such as AADL. A sub-space for Transaction abstraction for a SystemC-based approach could be the different 'untimed', 'loosely-timed' and 'timed' sub-spaces.
- RT Space: This space corresponds to the defacto notion of RTL (Register-Transfer Level). It contains components, entities, objects, models and languages such as VHDL, Verilog models. Corresponding sub-spaces could be gate-level and switch-level, which are otherwise considered as their own distinctive abstraction spaces in some contexts.
- C Space: This space contains components, entities, objects, models and languages that capture different accuracy/complexity trade-offs, e.g. Verilog-A, differential equations for circuit elements can be used to capture nanometric short-channel effects or simple linear V/I equations typically found in SPICE problems.
- P Space: This space contains components, entities, objects, models and languages such as SPICE systems for printed circuit boards and integrated systems, Maxwell electromagnetic solvers for physical design of radio frequency transceiver systems used in cellular networks and handheld devices, and thermodynamic systems solvers for the packaging of integrated high performance microcomputers.

The design of a typical programmable system-on-chip (SoC) based computer on a printed circuit board (PCB) in which the SoC is a monolithic hard processor IP and a field-programmable gate array (FPGA) fabric can comprise models and languages that cross different abstraction spaces. The PCB design extends over the C and P Spaces; hard and soft IPs extend over the RT to P Spaces; SystemC models for dataflow computation to be compiled to assembly and hardware accelerator using high-level synthesis extend over S/T and RT Spaces. Examples of specialised abstraction spaces for specific components are for computing component architectures instruction-set, micro-, and macro-architecture; whereas for networking components, OSI as in ISO/IEC 7498.

Foundational Construct 3. Categories, Functors, and Natural Transformations (Mathematics)
Category theory is a branch of mathematics that deals with abstract structures and relationships. The fundamental ideas in category theory are categories, functors, and natural transformations. By employing these concepts, category theory enables the formal description and analysis of relationships, hierarchies, and abstractions in different fields.

A **category** $\mathcal{C}$ is a mathematical structure consisting of:

- *Objects* ($\mathrm{Ob}(\mathcal{C})$)
- *Morphisms* between objects ($\mathrm{Hom}_{\mathcal{C}}(A, B)$)
- *Composition of morphisms*, which is associative
- *Identity morphisms* for each object, which serve as identity elements under composition.

A **functor** F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a map that:

- Associates to each object A in $\mathcal{C}$ an object $F(A)$ in $\mathcal{D}$
- Associates to each morphism $f : A \to B$ in $\mathcal{C}$ a morphism $F(f) : F(A) \to F(B)$ in $\mathcal{D}$ such that composition and identity morphisms are preserved.

A **natural transformation** η from a functor F to a functor G (both from $\mathcal{C}$ to $\mathcal{D}$) is a collection of morphisms in $\mathcal{D}$ such that:

- For every object A in $\mathcal{C}$, there is a morphism $\eta_A : F(A) \to G(A)$ in $\mathcal{D}$.
- For every morphism $f : A \to B$ in $\mathcal{C}$, the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{\eta_A} & G(A) \\ F(f) \downarrow & & \downarrow G(f) \\ F(B) & \xrightarrow{\eta_B} & G(B) \end{array}$$

A morphism $f : A \to B$ in a category is an **isomorphism** if there exists a morphism $g : B \to A$ such that $f \circ g = \mathrm{id}_B$ and $g \circ f = \mathrm{id}_A$.

The **Yoneda embedding** is a functor that embeds a category $\mathcal{C}$ into the category of functors from $\mathcal{C}$ to the category of sets, Set. Specifically, it associates to each object A in $\mathcal{C}$ the functor $\mathrm{Hom}(-, A)$, and to each morphism $f : A \to B$ in $\mathcal{C}$ the natural transformation $\mathrm{Hom}(-, f)$.

Employing the formalism of category theory allows us to delineate hierarchies and relationships within diverse computer language models, networks, and their associated abstractions with enhanced precision. This theoretical underpinning facilitates a deeper exploration into the connections between computer programs and algorithms articulated in formal languages, and between the structural elements of computer hardware architectures or software systems. Specifically:

- Computer programs and algorithms in formal languages:
  Through the lens of category theory, we can articulate the interplay between computer programs and algorithms expressed in various formal languages. Objects within this categorical framework might symbolise different programming or specification languages, with morphisms capturing translations or transformations amongst them. Composing morphisms elucidates the process of melding or modifying programs using diverse language constructs. This perspective furnishes a holistic yet rigorous means to investigate program transformations, language compatibility, and compositional attributes.
- Structural components of computer hardware and software systems:
  Similarly, category theory elucidates the intricate anatomy of computer hardware architectures and software systems. Here, objects could typify components or modules such as processors, memory units, or software modules. Morphisms might illustrate relationships like data flows, control signals, or function calls. This categorical insight refines our understanding of the structural dynamics within computer hardware or software, proving invaluable for their design, scrutiny, and enhancement.

Category theory's inherent aptitude for abstractly and systematically delineating hierarchical structures and relationships validates its utility for our objectives. Thus, we are prompted to harness the power of categorical reasoning in our exploration of programming languages, computer networks, algorithms, hardware architectures, and software systems, all pivotal in the design and cultivation of models for computing systems.

III. THE MODEL-OF-DESIGN FRAMEWORK

Having established the theoretical foundations for modelling languages, abstraction spaces, and category-theoretic structures, we now turn to detail the central elements of the model-of-design framework in Subsection III-A. Following this, in Subsections III-B and III-C, we will explore the pivotal properties and resulting corollaries derived from the framework.

A. CORE CONSTITUENTS

The model-of-design framework draws upon a range of components: specifications, architecture, implementation, evaluation, and design and development processes. In this section, we shed light on these concepts through individual definitions from Definition III-A1 to III-A4, culminating in the comprehensive model of design concept presented in Definition 13, as follows:

1) SPECIFICATIONS

Definition 1. Model of Specifications (MoS)
Let $\mathcal{C}$ be a category and $L = \langle \Sigma, G, S \rangle$ be a modelling language associated with an abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$. A *model of specifications* (MoS) within $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$ serves as a category of two categories - the functional and extra-functional specifications - and can be perceived as a modelling language.

1) **Categorical structure of MoS**:
   - Objects: Within the abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$, objects in MoS are constructed as pairs that combine a *model of functionality* (MoF) defined through the grammar G of L and a *model of extra-functional specifications* (MoX). Formally:

$$\mathrm{Ob}_{\mathbb{A}.\mathbb{S}_{\mathbb{A}}}(\mathrm{MoS}) : \langle \mathrm{MoF}, \mathrm{MoX} \rangle$$

   - Morphisms: Preserving the semantics S of L, morphisms in MoS express the transformations or relationships between the encapsulated objects. These transformations are contingent on the specific system's specifications, invariably aligned with the abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$.
2) **Linguistic structure of MoS**: A *language of model of specifications*, L_{MoS}, encapsulates:

- Σ_{MoS}: A finite set of strings representing functional and extra-functional specifications. These can include functional requirements (F_r), non-functional requirements (N_r), constraints (K_s), and conditions (C_s).
- G_{MoS}: A grammar:
  - α_{MoS}: Terminals symbolising specification primitives like computation and conditions.
  - V_{MoS}: Variables indicating intricate functional specification structures.
  - P_{MoS}: Production rules that transmute specification constructs into coherent design requirements.
  - S_{MoS}: The start variable.
- S_{MoS}: A semantic mapping from Σ_{MoS} ensuring clarity and interpretation.

The union of the MoF and MoX within MoS, in the categorical sense of the models, can be the merger of the models resulting in a superset of alphabets, grammar and semantics. The specifics of such operations can be model dependent.

Definition 2. Model of Functionality (MoF)
Given an abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$ described by the modelling language $L = \langle \Sigma, G, S \rangle$, a *model of functionality* (MoF) serves as a category of two categories (MoB and MoC) and can also be understood as a modelling language, detailing the interactions of all potential functional specifications of a system via the constructs, grammar, and semantics of L.

1) **Categorical structure of MoF**:
   - Objects: Within the abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$, objects in MoF are mappings that, via L, convert a set of input values and variable states (i, v) to outputs o. In terms of the abstraction space, this is articulated as:

     $$\mathrm{Ob}_{\mathbb{A}.\mathbb{S}_{\mathbb{A}}}(\mathrm{MoF}) : \langle i, v, o \rangle : i, v \mapsto o \text{with } G \in L$$

     Moreover, the objects amalgamate the *model of computation* (MoC) - elucidating computational structures and synchronisations - and the *model of behaviour* (MoB) - detailing internal dynamics of individual processes. Hence:

     $$\mathrm{Ob}_{\mathbb{A}.\mathbb{S}_{\mathbb{A}}}(\mathrm{MoF}) = \bigcup \langle \mathrm{MoC}, \mathrm{MoB} \rangle$$

   - Morphisms: Representing transformations or associations among these objects, morphisms in MoF align with the semantics S of L, contingent on the dynamics within the abstraction space $\mathbb{A}.\mathbb{S}_{\mathbb{A}}$.

2) **Linguistic structure of MoF**: The modelling language for MoF, denoted as L_{MoF}, is formally defined similar to the foundational structure of a modelling language. It is given by the tuple $L_{MoF} = \langle \Sigma_{MoF}, G_{MoF}, S_{MoF} \rangle$, where:
   - Σ_{MoF} is a finite set of strings formulated over an alphabet specific to MoF. These strings depict the functional specifications and are constructed in accordance with the grammar G_{MoF}. Each string in Σ_{MoF} symbolises a coherent functional sentence or construct of the system.
   - G_{MoF} is a grammar which specifies how functional constructs are organised. It can be denoted as $G_{MoF} = \langle \alpha_{MoF}, V_{MoF}, P_{MoF}, S_{MoF} \rangle$, where:
     - α_{MoF} comprises terminals or symbols that indicate primary functional elements.
     - V_{MoF} contains variables, apart from α_{MoF}, that perhaps signify more intricate functional constructs or dynamics.
     - P_{MoF} embraces production rules, enabling one to form functional descriptions from the variables and terminals.
     - $S_{MoF} \in V_{MoF}$ is the initial variable designating the beginning of functional descriptions.
   - S_{MoF} provides the semantic mapping for Σ_{MoF}, associating each functional specification with its pertinent meaning or context.

Such a linguistic structure aids in the comprehensive representation and understanding of system functionalities, covering everything from basic functional elements to more complex dynamics.

By representing the model of functionality as a category, we can leverage the formal framework of category theory to reason about the hierarchy, composition, and relationships between the channel interfaces, processes, and process networks. The category structure provides a coherent and rigorous way to analyse and understand the syntactic and semantic aspects of the computation model.

Remark 4 (Similarities between functionality and specification models). The functional model (MoF) can sometimes be equivalent to several notions of the term specification models e.g. [16] and [18], where they describe the specification model as the set of behaviours, channels and connectivity relations.

Definition 3. Model of Computation (MoC)
Within a category $\mathcal{C}$, a *model of computation* (MoC) bridges categorical and linguistic structures to elucidate computational abstractions:

1) **Categorical structure of MoC**:
   - **Objects**: Comprising I_p, P_p, and G_h, which denote channel interfaces, parameterisable processes, and hierarchical labelled directed graphs, respectively. These form the foundational constituents of computational paradigms.
   - **Morphisms**: Establish relationships between I_p, P_p, and G_h, according to the compositional rules intrinsic to the MoC, portraying the dynamic interconnections of computational systems.
2) **Linguistic structure of MoC**:
   - **Syntax**: Governed by the tuple $\langle \Sigma, G, S \rangle$, it characterises the grammatical structures and interrelations typifying computational constructs.
   - **Operational semantics**: Articulates the behavioural evolution of actors and processes, shaped by processes and signal transference. This encompasses rules that steer computational entities over temporal progressions. Incorporating the denotational insights of Lee and Sangiovanni-Vincentelli [4], processes in concurrent systems are represented as sets of potential behaviours. Composite processes yield behaviours intersecting those of individual components. Interactions arise via signals, collections of events described by value-tag pairs. When tags are totally ordered, they delineate timed models. Processes possessing identical tags exhibit synchronous signals.

Categorically, we can interpret the components of the tuple $\langle I_p, P_p, G_h \rangle$ within the framework of category theory:

1) I_p (language describing channel interfaces): In the category, I_p can be seen as the collection of objects representing the channel interfaces. Each interface corresponds to a specific input or output configuration of the computation model.
2) P_p (language describing parameterisable processes): The morphisms in the category can be associated with the processes described by P_p. The morphisms relate the inputs to the outputs, capturing the transformations performed by the processes.
3) G_h (set of labelled hierarchical directed graphs): The composition of morphisms within the category can be identified with the graphs in G_h. The composition rules and denotations define the process networks that interconnect the interfaces and processes, forming the structure of the computation model.
4) Parameters (X_p): The parameters in the model of computation can be linked to the additional variables and conditions within the category. These parameters define the initial conditions, status, and other operational variables, such as scenarios and modes.

The model of functionality (MoF) specifies what a system does. It outlines the functionality of the system without considering timing or other forms of behaviour. It is concerned primarily with the logical operations performed by the system and the data transformations these operations create. The model of computation (MoC), on the other hand, specifies how a system does what it does. The MoC dictates the execution semantics of a system, such as the rules for how operations can be ordered and how data can be exchanged. In other words, it sets the 'rules of the game' for computation and communication.

Example 3 (MoCs). Examples of MoC include: static data-flow (SDF), homogeneous (synchronous) data flow (HSDF), cyclo-static data flow (CSDF), boolean data flow (BDF), Dennis/dynamic data flow (DDF), variable rate dataflow (VRDF), multimode dataflow (MMDF), parametric synchronous data flow (PSDF), schedulable parametric dataflow (SPDF), scenario-aware dataflows (SADF), Kahn process networks (KPNs), non-determinate data flow (NDF), discrete events/time (DE/DT), synchronous computation (SY), continuous time computation (CT).

Remark 5 (On the graph components of MoCs). While models of computations do not include by definition graphs, our reasoning for including graphs stems from two points:

- as demonstrated by the tagged-signal framework for comparing models of computations by Lee and Sangiovanni-Vincentelli [4], models of computation can be largely captured by signals, processes and their relations to the tag system. Since processes can be used as vertices and signals can be used as edges. This makes graphs implied from most MoCs.
- as a matter of convenience, since many models of computations such as Matlab/Simulink and SCCharts* [19] are graphically oriented at least from a design-entry point of view, it is deemed appropriate to include graphs as part of the MoC definition.

Remark 6 (Relation to MoCs). Models of computation [20] typically defines 1) a network of interconnected set of computing elements referred to as processes (actors, kernels or tasks), including the rules of composition, 2) the operational semantics or conditions and rules for firing or execution of processes or the conditions 3) initial and status for the conditions for the channels, process configurations and mode/scenarios. MoCs are often represented as set of input/outputs; ports and channels (interfaces, I); set of process (processes, P); process network (graph, G). Consequently, the MoCs are defined as the language describing the processes, interfaces and the graph in addition to the rules of composition, operational semantics, and other emerging properties. The initial and status conditions of the processes and interfaces map to the parameters. In the categorical view, we can represent a MoC as a category representing a model of computation consists of the following components:

- Objects: The objects in the category represent different configurations or states of the computation model. Each object corresponds to a specific arrangement or setup of the computing elements, such as processes, actors, kernels, or tasks.
- Morphisms: The morphisms in the category represent the transformations or mappings between different configurations or states of the model. Each morphism captures the transition or change from one configuration to another, reflecting the behaviour and interactions of the computing elements.
- Composition: The composition of morphisms defines how transformations can be combined or sequenced. Given two morphisms, $f : A \rightarrow B$ and $g : B \rightarrow C$, their composition denoted as $g \circ f : A \rightarrow C$ represents the transformation obtained by applying f followed by g. The composition allows for the interconnection and sequential execution of processes within the computation model.
- Associativity Law: The composition of morphisms is associative, meaning that for any three morphisms $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$, the composition is associative as $(h \circ g) \circ f = h \circ (g \circ f)$. This law ensures that the order of composition is irrelevant and allows for the chaining of transformations.
- Identity Morphisms: For every object A, there exists an identity morphism $id_A : A \rightarrow A$ that serves as the neutral element with respect to composition. The identity morphism preserves the configuration or state of the object, leaving it unchanged when composed with other morphisms. The identity morphism ensures that every object has an identity element and provides a consistent starting point for composition.

The category structure, with its objects, morphisms, composition, associativity law, and identity morphisms, allows us to formally reason about the behaviour and transformations within the model of computation. It provides a framework for studying the properties, relationships, and compositional aspects of the MoCs, their interactions, and the overall computation process.

Remark 7 (MoCs hierarchies). Basic models of computations, thanks to their formalism, can be typically defined and mapped [21] in the order of expressiveness and analysability including: homogeneous synchronous dataflow and marked graphs (HSDF/MG), synchronous dataflow and weighted marked graphs (SDF/WMG), computation graphs (CG), cyclo-static dataflows (CSDF), parameterised synchronous dataflow (PSDF), variable rate dataflow (VRDF), finite state machine scenario aware dataflow or heterochronous dataflow (FSM-SADF)/HDF, variable phase dataflow (VPDF), boolean dataflow (BDF), SADF, Kahn process networks (KPN), dynamic/Denis dataflow (DDF) and reactive process networks (RPN). On the basis of the relation between these basic MoCs, different transformational relationships can be drawn to reason on the equivalency and gaps between the level of details each MoC has. Since our general definition of MoCs includes general constructs such as processes, interfaces and graphs; in addition to the syntax and semantics of these constructs, we can reason about different classes of MoCs and provide execution frameworks to them.

Remark 8 (The association with the tagged-signal model). The model of computation defined here can map to the tagged-signal model [4], when the parameters and the operational semantics of the processes and interfaces of MoC are related to the tagged-signal model for the signals and processes. The processes in our MoC maps to the processes and the interfaces in our MoC map to the signals in the tagged signal models. When defining values and tags as parameters of the processes and interfaces, we map the MoC to the tagged signal model. Henceforth, by defining the relation of the tags, different variations of the MoC can be defined.

Remark 9 (The suitability of MoCs to abstraction spaces). While MoCs are quite general, some MoCs fit abstraction spaces better than others due to their level of details, especially concerning time, e.g. KPNs and SDFs family fits transaction levels better; synchronous model fits RT space abstraction space better; continuous time model fits circuit and physical abstraction spaces better. In computer science, MoCs concepts may be included here, but we think they map better within architectures of computers as shown in Example 6.

Remark 10 (The role of behaviour in MoCs). Traditionally, models of computation such as in MMDF, KPN or SADF, have been abstract about what are the limited set of possible (functional/ logical/ arithmetic/ relational) behaviours within the processes. This lack of explicitness renders some MoCs, on their own, insufficient to reason about functional correctness of refinement, synthesis, transformation, or compilation; therefore we add the concept of the model of behaviour to complement such a lack.

Definition 4. Model of Behaviour (MoB)
Let $\mathcal{C}$ be a category where:

1) **Objects** are computational behaviours. Within the category $\mathbb{B}$, an object symbolises the assignment of an input i to a behaviour, formalised as:

$$\text{Ob(MoB)} : \lambda x.(B) \text{ where } x \in i,\ B \in \mathbb{B}$$

In a manner analogous to programming functions, this object denotes the binding of an input i to a variable x, subsequently replacing x in the behaviour B with i.

2) **Morphisms** describe transformations between behaviours. These transformations are analogous to the function mappings in standard programming,

described as:

$$\mathrm{Hom}_{\mathrm{MoB}}(A, B) : f(i, x, o) : i \mapsto o, x = B \in \mathbb{B}$$

Here, both A and B belong to $\mathbb{B}$. The symbols i and o denote input and output, respectively.

Thus, a *model of behaviour* (MoB) within $\mathcal{C}$ offers a categorical representation of behavioural constructs, grounding itself in principles reminiscent of the λ-calculus. This representation is both rigorous in theory and directly relatable to traditional programming constructs.

Remark 11 (On the choice of λ-calculus for MoB). As it is noted that λ-calculus is a well-known fundamental notion for defining (mathematical) functions within the theory of computation, it is deemed appropriate to represent (behaviours of) computation in a fundamental way. This definition additionally allows capturing analogue, mechanical and physical behaviours that can be interesting for capturing information processing associated with radio engineering, optical systems and electro-mechanical systems.

Remark 12 (MoC and MoB).

- The explicit definition for the set of behaviours makes it possible to allow unambiguous execution of functional specifications in the sense of hardware/software codesign and enables the expressiveness of MoCs to model concepts such as instruction set architecture (ISA), and the arithmetic and logical operations within a programming language. For example, the modelling framework within ForSyDe [18] uses MoCs in addition to the languages Haskell and SystemC to enable clear and explicit notion of behaviours and computation.
- MoB can be considered as a complementary concept to the MoC or as a superior concept that subsumes MoC or as a subset of MoC depending on how the MoC and MoB are defined.

MoF captures the functionality of the system including (possibly) model of computation (MoC) as defined extensively in literature [4] and (possibly) model of behaviour (MoB). MoF and MoB map to the behavioural domain of Gajski-Kuhn Y-chart and computation independent model (CIM) in OMG's model-driven-architecture (MDA) terminology. The model of behaviour refers to the syntactic and semantics used to capture the computation within each process/actor. It is however necessary to distinguish our notion of MoF from non-computational but sometimes considered within the functional requirements as for example is the case of avionics ARP4754 which falls within what we call, eXtra-functional, i.e. MoX. In fact, most of the industrial standards such as ISO262626 and IEC61058, on specifications and compliance can be mapped to MoX (Definition 5) and the associated design rules (Definition 9).

Example 4 (MoF, MoC and MoB). Examples to illustrate possible definitions for MoF, MoC and MoB may include:

- Examples of *MoC* are synchronous model of computation and scenario-aware dataflow model of computation. An example of *MoB* is functional language Haskell that defines language for behaviours. A resulting *MoF* for the mentioned MoC $\cup$ MoB is ForSyDe modelling framework.
- VHDL (simulation subset) can be described as *MoF* with *MoC* being the discrete time model of computation, while the *MoB* being the valid statements and expressions within the processes/ procedures/ functions/ modules. In HDL where behavioural style is used, the MoC becomes the synchronous MoC. Synopsis and Cadence design frameworks are prime examples for use of such MoFs.
- Simulink and AMD/Xilinx model composer can be considered as computer-aided design tools that utilise Matlab language (M files) and the associated execution models. In this case, the *MoF* is the (synthesisable subset of) Matlab language with *MoC* being a discrete event/time model of computation, while the *MoB* being the valid expressions/computations that can be done within the individual Simulink blocks or callback functions.

Remark 13 (The Overlap Within MoF). Several standard design specifications, such as IEEE 754 for floating point within the simulation subset of design specification languages like IEEE 1076/1364/1666/1800 (VHDL/Verilog/ SystemC/System Verilog), can sufficiently describe models for the grammar (syntax) and semantics of the functional/behavioural specifications. These languages implicitly contain or model various models of computation such as discrete time/ events for digital systems, and continuous time for analogue and mixed-signal systems. In practice, the (implied) functional, behavioural and computational models coexist and merge within the same language. Furthermore, system designers can include in the functional specification other notions that we consider, for theoretical purposes, to be included in what we call extra-functional specification, e.g. constraints on timing.

Remark 14 (Use of Categories for MoF, MoC and MoB). Category theory provides a framework to describe and analyse different abstract objects. This can also be extended models of computation, behaviours and functionalities, and their relationships. By representing models as categories and defining appropriate morphisms between them, we can study their properties, transformations, and connections. Here is how category theory can be used to describe different models and their relationships:

1) Category for a Model of Computation (in general): To describe a category as a model of computation, we define the category as follows:
   - Objects: Objects in the category represent different processes and channels.

- Morphisms: Morphisms in the category represent transformations or mappings between the processes, interfaces and graphs. These morphisms capture the relationships, translations, or compositions between different objects.

2) Category of Models of Computations (in general): To describe the category of models of computations, we define the category as follows:
   - Objects: Objects in the category represent different models of computation.
   - Morphisms: Morphisms in the category represent relationships or connections between different models of computation. These morphisms capture the mappings or transformations between models.
3) Category for a Model of Computation (specific): Let us consider the specific model of computation called the 'synchronous data flow (SDF)' model. We can define a category where:
   - Object: The object represents the SDF models.
   - Morphisms: Morphisms in the category represent transformations or mappings between different SDF models.

     The category structure allows us to reason about the properties, transformations, and relationships specific to the SDF model. We can study the composability of SDF models, the existence of isomorphisms or equivalences, and other categorical constructions that capture the essence of the SDF model.
4) Category for Models of Computation (specific): Let us consider a specific collection of models of computation, including SDF, HSDF, and CSDF. We can define a category where:
   - Objects: Objects in the category represent different models of computation such as SDF, HSDF, and CSDF.
   - Morphisms: Morphisms in the category represent relationships or connections between these models, capturing the mappings or transformations between them.
   - Composition: Composition of morphisms represents the composition of transformations or mappings between different models of computation.

     In this category, we can study the relationships, similarities, and differences between SDF, HSDF, and CSDF models. We can analyse the morphisms between these models, investigate the compositionality and composability properties, and explore categorical constructions that capture the interactions and transformations within this collection of models.

   Note: In all the provided examples, it is assumed that each object possesses an identity morphism, signifying the identity transformation or mapping of an object onto itself. Additionally, we take for granted that both the associativity and identity laws are upheld. This means the composition of morphisms adheres to the associative law, and the identity morphisms comply with the identity laws.

To summarise, using category theory to describe models of functionality (MoF), models of computation (MoC), and models of behaviour (MoB) presents several valuable insights and advantages:

1) *Universality*: One of the core benefits of using category theory in these contexts is its ability to provide a universal language for mathematics and computer science. This allows for the encoding of different computational and behavioural models within a unified framework, facilitating comparison and interaction between different systems and models.
2) *Structural Insights*: Categories can highlight the structural properties of MoF, MoC, and MoB. The morphisms (arrows) in a category represent transformations or relationships, revealing structural insights about the entities being modelled. These structural properties can provide a deeper understanding of the system, and can often be used to identify common patterns or structures across different systems.
3) *Abstraction and Generality*: Category theory provides a high level of abstraction. This means that the specifics of individual objects within a category can be abstracted away to focus on the relationships between them (the morphisms). In the context of MoF, MoC, and MoB, this could help to identify commonalities and differences at a high level, without getting bogged down in the specifics of each individual model.
4) *Functors and Natural Transformations*: Category theory introduces the concepts of functors and natural transformations. Functors are mappings between categories that preserve their structure, while natural transformations are mappings between functors that preserve their structure. In the context of MoF, MoC, and MoB, functors could be used to translate between different models or representations, while natural transformations could represent higher-level transformations or modifications.
5) *Compositionality*: The compositional nature of category theory, where morphisms can be composed to form new morphisms, is highly relevant in the context of MoF, MoC, and MoB. This can model the composition of functions or behaviours in a system, and reflects the compositional nature of software and systems design.
6) *Yoneda Embedding*: The Yoneda embedding, a concept in category theory, says that a category can be fully embedded (i.e., faithfully represented) within a category of functors defined on it. This gives a powerful way to represent and work with the category, and in the context of MoF, MoC, and MoB, could provide a new perspective or approach to these models.

Definition 5. Model of Extra-Functional Specifications (MoX)
Let $\mathcal{M}$ be a category that encapsulates the extra-functional characteristics of a computational system within a given abstraction space. A *model of extra-functional specifications* is hereby defined by the category $\mathcal{M}$:

- *Objects:* Constituting the set χ, these objects represent individual extra-functional variables, each demarcating distinct facets of non-functional attributes inherent to the computational system.
- *Morphisms:* Represented by functions, these capture the dynamic interplay and transformative relationships between the extra-functional variables. Specifically, for any pair of variables $\chi_1, \chi_2 \in \chi$, the collection of morphisms is denoted as $\mathrm{Hom}_{\mathcal{M}}(\chi_1, \chi_2)$.

Within this categorical framework, the extra-functional variables coalesce to define two distinct, yet interrelated, aspects:

- The environmental preconditions under which the design operates, denoted as ϑ.
- The constraints and regulations imposed upon the design, either holistically or partially, represented by ϱ.

As a synthesis, the category $\mathcal{M}$ is inherently associated with this ensemble of conditions and constraints, articulated as: $\mathrm{MoX}(\chi) : \vartheta \cup \varrho$.

With this categorical representation, the 'model of extra-functional specifications' can be understood as the category $\mathcal{M}$, where the objects are the extra-functional variables χ and the morphisms represent the expressions operating on these variables. The expressions collectively describe the extra-functional specifications that the design implementation needs to consider, encompassing both the ambient/environmental conditions ϑ and the design constraints/rules ϱ applicable to the entire or part of the design. To explain this further, consider a system where we have functional specifications depicted by a model of functionality (MoF) and extra-functional specifications defined by a model of extra-functional specifications (MoX). These are captured as objects in the model of specifications (MoS), formalised as the pair $\langle \mathrm{MoF}, \mathrm{MoX} \rangle$.

To express the application of extra-functional specifications on specific parts of functionality, one could conceive the model of extra-functional specifications (MoX) functor that maps from the category of the model of functionality (MoF) to the category of models of specifications (MoS). This functor can be defined to selectively apply the extra-functional specifications to the functional model.

Let $\mathrm{Spec} : \mathrm{MoF} \to \mathrm{MoS}$ denote this functor, which maps an object f in MoF to an object $\mathrm{Spec}(f) = \langle f, \mathrm{MoX} \rangle$ in MoS. This functor can also map a morphism $g : f_1 \to f_2$ in MoF to a morphism $\mathrm{Spec}(g) : \mathrm{Spec}(f_1) \to \mathrm{Spec}(f_2)$ in MoS, thereby capturing the relationships or transformations between different models of specifications.

The definition of this functor allows extra-functional specifications to be applied selectively to specific parts of the functionality in a rigorous, category-theoretic manner, thus maintaining soundness and integrity of the whole system's specification.

In a less formal language, we are taking our functional specifications (the bits that tell us what the system does) and our extra-functional specifications (the bits that tell us under what conditions the system operates and what constraints it needs to satisfy). We then combine these specifications to form a complete model of the system's specifications. This process of combining can be selective – we can choose to apply certain extra-functional specifications to particular parts of the functionality. The mathematics of category theory gives us a robust and formal way to do this, ensuring that our model remains consistent and coherent.

Remark 15 (Considerations for the scope of MoX).

- Extra-functional specifications can comprise environmental constraints such as the input stimuli and output loading/connections in addition to external factors that affect the system/design operation. This includes operating conditions that affect the evaluation of the system performance such as ambient conditions within the design and its settings that affect the design performance/cost.
- Extra-functional specifications can describe additional constraints that restrict the functional or architectural or implementation constraints. This is useful to further restrict the functional model or architectural or implementation models.
- Extra-functional specifications can describe physical/business/mechanical characteristics and intents that are needed or desired for the design. Some industrial application domains include these aspects in system specifications or requirements.

Remark 16 (On Time-Aware Programming Languages). Some programming languages, such as SystemC, the simulation subset of VHDL, synchronous languages like Esterel, and the programming languages ISO/IEC 8652/9899 ADA/C with real-time extensions (accompanied by the Ravenscar profile or real-time operating systems), incorporate explicit time constructs like `wait time seconds`. Such languages extend beyond purely functional models because they incorporate awareness of temporal dimensions of the 'real world' independent of the computing machines.

In addition to temporal aspects, there are properties from the real world, such as energy dissipation, temperature, and other physical events in cyber-physical systems, which are not inherent aspects of computation per se, but arise when implementing computation on tangible hardware. These extra-functional aspects are encapsulated within extra-functional models as they deal with elements external to the (pure) function of the computation.

Consequently, programming languages with spatio-temporal awareness offer more than 'just' functionality; they serve as specification models incorporating both functional and extra-functional properties. This makes them particularly adept at modelling real-world systems where both the computations (functional aspects) and the conditions of their implementation (extra-functional aspects) matter.

Example 5 (MoX).

- System abstraction space: An MoX can comprise requirements for χ_i for low-power design minimising average power consumption for component x and χ_j specification regarding throughput per application, θ_a such that it is not below a specific threshold Θ as follows:

$$\chi_i : min(P(x)) \underset{x\in(x_i,x_j]}{\arg\min} P(x) = \{x \mid P(x) = \min_{x'} P(x')\}$$

$$\chi_j : \underset{\forall x\in\bigcup a}{\arg\max}\, \theta(x),\ \theta(x) > \Theta_x$$

  And an extra-functional specification, χ_k, describing the junction temperature is the temperature allowed at the processor die μC to be designed for -55 to 125° C.

$$\chi_k : -55^\circ < T(\mu C_x) < 125^\circ$$

  An example of system, S that shall be satisfying or complying to a set of safety rules described in industrial standards OSI26262/ ARP4754A/ ARINC653 /IEC61508 donated by Λ, can be described as: $S \vdash \Lambda$

- RT abstraction space: Synopsis design constraints (SDC) and unified power format (UPF,IEEE1801) can be considered examples of models used to caputure extra-functional specification exemplified by timing/performance, power/energy, and area (PPA) for RTL. Such constraints restrict the implementation of the functional/behavioural models and can impose extra requirements on the behaviour, architecture, implementation of the system, and the design rules used. This includes for example clock/area/power (timing/ synchronisation) constraints, input/output delay/load constraints, environmental constraints, design rules constraints, technology constraints. The MoX can further be per-component constraints for the architecture/ implementation or at the design/system level constraints concerning turn-around time or accuracy or operating conditions. Examples include: environmental (MoX): set_operating_conditions, set_load/drive/driving_cell/ fanout_load input_transition/ port_fanout_number, design rules (Λ), set_max_capacitance/ fanout/ transition, timing: wireload models set_wire_load_mode/ model/ selection_group, create_clock /generated_clock, set_clock _latency/ transition, clock_uncertainty input/ output_delay, power: set_max_leakage /dynamic_power

More examples for how to specify extra-functional properties formally are in [22] and [23].

2) ARCHITECTURE

Definition 6. Model of Architecture (MoA)
A *model of architecture* $\mathcal{A}$ is a category formalised to represent the architectural aspects of computational systems within a designated abstraction space. The constituents of $\mathcal{A}$ are delineated as:

- *Objects:* Each object within $\mathcal{A}$ is denoted by the tuple $\langle I_p, C_p\rangle$. Here, I_p characterises interfaces, which depending on the abstraction level, may be understood as ports (in SL abstraction), pins (in RT abstraction), nodes (in C abstraction), or points (in P abstraction). Conversely, C_p encapsulates parameterisable components, either hardware or software in nature, interlinked by interfaces C_i, C_o such that $C_i, C_o \in I_p$.
- *Morphisms:* These capture the interactive dynamics between architectural elements. Specifically, they are represented by G_c, a collection of labelled directed graphs. The edges within each graph, $C_p \times I_p \cup I_p \times C_p$, are governed by established compositional rules and denotations, ensuring the fidelity of both the syntax and semantics of the architecture's representation.

Expressed succinctly, the model of architecture $\mathcal{A}$ can be described by the tuple:

$$\mathcal{A} : \langle I_p, C_p, G_c\rangle$$

A *language of model of architecture*, denoted as L_{MoA}, is a modelling language given by the tuple $L_{MoA} = \langle \Sigma_{MoA}, G_{MoA}, S_{MoA}\rangle$, where:

- Σ_{MoA} is a finite set of strings formed over an alphabet which encapsulates the architectural constructs. Each string in Σ_{MoA} represents a well-formed architectural description, which may include constructs such as interfaces (represented as I_p), parameterisable components (C_p), and relationships (given by the labelled directed graphs G_c).
- G_{MoA} is a grammar defined as $G_{MoA} = \langle \alpha_{MoA}, V_{MoA}, P_{MoA}, S_{MoA}\rangle$, where:
  - α_{MoA} is a finite set of terminals or symbols that represents architectural primitives such as ports, pins, nodes, or points.
  - V_{MoA} is a finite set of variables distinct from α_{MoA} that might denote more complex architectural structures.
  - P_{MoA} is a set of production rules, translating architectural constructs into meaningful design descriptions.
  - $S_{MoA} \in V_{MoA}$ is the start variable that initiates the architectural descriptions.
- S_{MoA} provides a mapping from Σ_{MoA} to the semantics associated with each architectural description, thereby attributing meaning and context to the representations in Σ_{MoA}.

In this category, the subscript $_p$ indicates possible parameters. G_c denotes that components can be hierarchical, nesting other components and/or MoAs. The model of architecture aligns with the structural domain within the Gajski-Kuhn Y-Chart and resides within the abstraction spaces $\mathbb{A}$, as outlined in Foundational Construct 2. This corresponds to the platform-independent models (PIMs) in the OMG's model-driven architecture (MDA) terminology at the system or transaction-level abstraction space and to the technology-independent generic logic libraries in logic synthesis at the RT. MoA acts as a bridge, easing transitions from the S/T Space to the RT Space, or from the RT Space to the C Space. The architectural model brings forth concepts such as meet-in-the-middle and platform-based design, model-driven architecture, and more. Architectural elements and design are pervasive in embedded hardware and software industrial sectors, as evidenced in standards like DO-178C for avionics and AutoSAR for automotive.

Remark 17 (MoAs and MoFs).

- Architectures, in the broadest sense, are employed to describe abstract relations between hardware elements, as seen in computer organisation and architecture (more suited for MoAs). Alternatively, they depict software modules, classes, or entities and their interfaces in software engineering, akin to software architecture (more apt for MoFs).
- The distinction between MoA and MoF blurs when considering the instruction set architecture (ISA) where hardware and software intermingle. From a hardware standpoint, instructions are decoded in the instruction decoding logic within computer architecture; hence, the instruction set can be part of the computer's architectural MoA. From a functionality perspective, ISA indicates the range of functional behaviours that encapsulate the system's functionality, placing it within MoB ($\in$ MoF).

Remark 18 (The Role of Parameters in MoA). The parameterised nature of the model of architecture facilitates the configuration of architectural components across abstraction spaces. By deliberately incorporating parameters into the architectural model, various configuration and mode parameters for both software and hardware components and interfaces can be defined across different abstraction levels. This is advantageous in situations where parameters enable diverse design trade-offs both vertically (across varying complexity levels) and horizontally (based on component and interface choices). For instance, processor architectures can encompass different instruction sets and micro-architectures. Operating systems can manifest in varied configurations, customisations, builds, and architectures. Parameters elucidate why identical architectures with differing settings yield diverse platforms. These parameters also resonate with the notion of platform/system configuration.

Remark 19 (Topologies and Interfaces in MoA). By incorporating interfaces and graphs directly, we aim to highlight the architecture's topologies and interconnections. Topologies might be depicted through hierarchical graphs such as daisy-chains, stars, trees, meshes, and toruses, whilst interfaces can capture hardware/software component interactions. Examples include the OMG's interface definition/description language (IDL) [24], [25], application program/binary interfaces (API/ABI), and various interconnection protocols. Across different abstraction spaces, topologies can transform their meanings; for instance, in the system space, topologies might represent logical connections or data flow between subsystems, whereas at the physical space, they might focus on interfaces and isolation, considering factors such as signal integrity or 3D and 2.5D IC packaging.

Remark 20 (Time-triggered architectures (TTA)). Time triggered (and spatiotemporal in general) architectures combine the connectivity of components with the explicit notion of time (and space). Time (and spatiotemporal) architectures are special cases of our notion of MoA at the physical space. Generally, TTAs do not match MoA in other spaces directly, as MoA attempts to remain general enough to encompass implementation models across various levels of abstraction that could be virtual, emulated, or software-intensive; which does not necessarily require explicit notions of space/time/energy; or at least not in a consistent manner. See also Remark 21 for further insights.

Remark 21 (MoA and evaluation models). The inclusion of extra-functional related properties in architectures can be useful, but in our work, we attempt to separate the architectural concerns/aspects (MoA) from the evaluation concerns/aspects (see Definition III-A4 on model of evaluation, MoE). In principle, we view architectures as technology-independent and platform-independent while useful evaluation metrics should be technology-dependent and platform-specific, hence We argue that evaluation should be applied to the implementation and not to the architecture.

Remark 22 (On the behaviour, interaction and priority (BIP) framework). The BIP framework [26], is used to reason about compositionality using a component-based design/construction approach. As the framework primarily defines atomic components and glue operators that can be used to compose other complex components comprising the system; the framework can be perceived to be compliant with our notion of MoA, as atomic/composite components can map to our notion of component, glue operators can be subsumed within our notion of categorical functor or graph in addition to the overall grammar (syntax) and semantics of the model.

Example 6 (Architectural components and interfaces).

- At system space: hardware and software typically possess distinct architectures. When integrated, they form a comprehensive system architecture:
  - A common hardware architecture encompasses processor cores, which may include cache systems from vendors like Intel, AMD, and ARM; memory storage such as NAND flash, scratchpads, DDR DRAMs, and SRAMs; I/O device adaptors suitable for displays,

audio, GPS, motion actuators, and instrumentation sensors; and both on-chip and off-chip interconnects, examples being PCIe, NoCs, various AMBA fabrics, Intel Avalon fabric, SerDes, USB variants, SDIO/I2C, SPI, UART, eMMC, and a range of networking protocols.
  - A typical software architecture comprises board support packages (BSP) and firmware/BIOS; device drivers for USB, Display, and other I/O; kernels for OS, RTOS or hypervisor, incorporating scheduler, process managers, memory managers, and other essential functionalities; and application programming/binary interfaces (API/ABI). For automotive applications, EAST-ADL and AutoSAR may represent the *MoA* architecture at the system space for software.
- At the RT abstraction space: Generic gate-level libraries, described in a structural style of Verilog, cater to nets, cells, pins, ports, and clocks for logic synthesis. These standard-cell libraries can be considered as models of architecture for many logic computer-aided design (CAD) synthesis tools. On the other hand, IP XACT (IEEE 1685/ IEC 62014) is viewed as an MoA language. With such architectural languages, architectures can be refined for various features, including testability, power management, and reconfigurability.
- At the circuit space: Custom integrated circuit (IC) models use circuit components like transistors, metal vias, and interconnects. These models, described in 'library exchange format (LEF)', are predominant examples of architectural component models employed in custom IC flows such as Cadence Virtuoso.

Utilising these fundamental constructs of components, interfaces, and graphs facilitates the creation of diverse hardware/software architectures, ranging from transistor scale to supercomputing neural networks. For instance, a typical bus-based MPSoC architecture designed for custom heterogeneous shared-memory could involve components such as Caches, RAMs, and multi-threaded CPU decoders, among others. By integrating 2-D NoC graphs of these training nodes and low-latency switches, expansive architectures like the 'Dojo' can be formulated. Furthermore, the open systems interconnection (OSI) for networks and Flynn's taxonomy for computer architecture [27] can be viewed as subsets of the classification of *MoA* for computers and networks. A reduced instruction set computer (RISC) stands as an example of a *MoA* across multiple levels of abstraction. Moreover, in computer science, certain concepts, from circuits in terms of logic gates to Turing machines, are classified under architectures as they describe the grammar and semantics of computational machines.

Example 7 (Relation of computing machines to models of architecture and functionality). Computing machines, such as finite state machines, push-down automata, and abstract computing machines, typically comprise a state (register), a head (encompassing a memory input/output controller, program pointer, and logic unit), and a tape (memory) containing possible instructions and data (program and data). In this context, if we consider the computing machine without the actual content of the tape, it becomes a model of architecture, representing only components and their operational semantics. However, when given specific tape content, the machine embodies a model of functionality or behaviour upon a model of architecture, where the tape's content (program) signifies a subset of potential behaviours or functionalities.

To illustrate this relation, consider a Turing computing machine, TM, defined as the tuple $\langle Q, q0, L, b, \Sigma, \delta, F\rangle$, comprising:

- A set L of symbols that TM's tapes can hold. We assume that L includes a specific "blank" symbol, denoted by b, a "start" symbol, represented as S, and various other symbols. We refer to L as TM's alphabet.
- $\Sigma \subseteq L - \{b\}$ represents the input symbols permitted on the initial tape content.
- A set Q of potential states for TM's register. It is assumed that Q contains a specific start state, $q0 \in Q$, and a halting state, qhalt $\in Q$. $F \subseteq Q$, represents the final states (or accepting states) with $F = \{qhalt\}$.
- A function $\delta : (Q-F)\times L \mapsto Q\times L\times\{l, S, r\}$ delineates the rule by which TM operates at each step. Known as TM's transition function, the head can move Left (l), Right (r), or Stay in place (S). The machine halts if the transition function is undefined for the current state and symbol on the tape. If the machine attempts to move left from the tape's leftmost position, it remains stationary.

From this example, the TM model can be interpreted as:

1) An architecture comprising components: a tape (memory) linked to a head (a memory controller connected to a program counter) at a specific position, with the transition function acting as the control unit with an instruction decoding unit.
2) By suitably encoding the symbols in the tape (L) to represent computation, the model, when viewed through the contents populated on the tape, can primarily be perceived as a model of functionality. The potential content sequences encoded by the tape can correspond to behavioural computations. The associated operational semantics is the model of functionality (MoF) comprising one process. The machine, minus the tape, simply provides the operational semantics describing the process output generation.
3) A third perspective suggests that the machine, with designated contents on the tape, symbolises a functional behaviour mapped onto architectural components (MoF x MoA).

3) IMPLEMENTATION

A representation of an implementation (implementation model) is the result of partial or complete design process.

The implementation model is not the same thing as the (physical) implementation, as it is often that some additional manufacturing or (virtual) prototyping or deployment effort can be needed to realise the implementation model. This is common for integrated circuits and systems-on-chips manufacturing; discrete circuit assembly on printed circuit boards or racks on modules for electronic controllers units; or burning-in device images/binaries on field-programmable gate arrays or processor-based emulation systems and virtual prototypes. Before we can move further in the definitions, we explain the concepts: refinement and abstraction as general operators on models; in addition to design rules.

Definition 7. Refinement and Abstraction
Refinement: Let $\mathcal{M}_{\tau.e}^{A.s}$ be a model within a model of design (MoD). Refinement is a functor that takes $\mathcal{M}_{\tau.e}^{A.s}$ and transforms it into a new model $\mathcal{M}'_{\tau.e}{}^{A+i.s+j}$, for any $i, j \in \mathbb{N}_0$ such that $i + j > 0$. This process of refinement adds further detail to the model, progressing from higher to lower levels of abstraction. *Abstraction*: Let $\mathcal{M}_{\tau.e}^{A.s}$ be a model within a model of design (MoD). Abstraction is a functor that transforms $\mathcal{M}_{\tau.e}^{A.s}$ into a new model $\mathcal{M}'_{\tau.e}{}^{A-i.s-j}$, for any $i, j \in \mathbb{N}_0$ such that $i+j > 0$. Through abstraction, details are removed from the model, leading to a higher level of abstraction.

Example 8 (Refinement and Abstraction). Refinement and abstraction comprise various transformations, inferences, or design steps that alter the level of abstraction in models. Examples may include elaborating design specifications to unfold hierarchies, spatially mapping an abstract behavioural specification to more detailed architectural processing elements, or transforming specification formats from a more abstract language to a more detailed one (compilation and synthesis).

Definition 8. Design Decisions (Δ)
Let $\mathcal{M} = \langle \text{MoS}, \text{MoA} \rangle$ be a source category where objects represent either models of specification or models of architecture, and morphisms denote transformations between these models. Let $\mathcal{I}$ be the target category with objects representing models of implementation. *Design decisions*, denoted as Δ, serve as a functor:

$$\Delta : \mathcal{M} \to \mathcal{I}$$

This functor maps:

1) Objects in $\mathcal{M}$ to objects in $\mathcal{I}$, translating or refining specifications and architectures into implementations.
2) Morphisms in $\mathcal{M}$ to morphisms in $\mathcal{I}$, transmuting the transformations in source models into corresponding transformations in target implementations.

Design decisions encapsulate an assembly of decision-making algorithms, frameworks, and actions. These decisions articulate diverse actions like architectural selection, hierarchy elaboration, topology configuration, component inference, routing, resource allocation, partitioning, mapping, scheduling, configuration, planning, and variable dimensioning. Central to this definition, design decisions can be perceived as a linguistic structure, offering syntax and semantics for capturing, conveying, and executing decisions. This linguistic view underpins the process of refining abstract models into tangible implementations.

The categorical interpretation of design decisions establishes a category functor $\mathcal{C}$ between categorical objects corresponding to different models or refined models of specifications, architectures, or implementations. The morphisms in $\mathcal{C}$ embody the decision-problem algorithms transforming an input model into an output model, reflecting the transformation or refinement process.

Example 9 (Design Decisions).
- Examples of design decisions at the system space may include: processes to processing core mapping, infinite channels synthesis to finite point-to-point buffers or multi-secondary multi-primary interconnect fabrics and packet-switching networks, instruction-set selection for behavioural synthesis, processor architecture selection, cache and memory sizing, memory hierarchy selection, application to virtual partition mapping (e.g. in aerospace ARINC standards), partition scheduling, runnable to tasks mapping (e.g. in automotive standards such as AutoSAR), scheduling algorithm selection (e.g. in real-time systems such as preemptive scheduling, earliest-deadline first, etc.), networks topology selection, time-division multiplexing sizing (e.g. in TDMA-based cross-bars and network-on-chips or in AFDX avionics network), switching mechanism selection.
- Examples of design decisions at the RT space of abstraction: for architectural designing, i.e. moving from specification to a generic technology-independent architecture, design decisions include: architecture selection for arithmetic operators, logic pruning, carry-save arithmetic, constant propagation, logic speculation, resource sharing, (non-functional) redundancy removal (especially when specific reliability is not an extra-functional requirement), finite state machine control logic encoding, multiplexer/arithmetic optimisation, retiming, clock gating. Design decision for implementing generic architecture into technology-specific implementation may include: clock-tree synthesis, floor-planning and wire routing, multi-bit merging, multi-voltage multi threshold-voltage scaling for delay and leaking-power minimisation, multi-mode multi-corner optimisation,

power shut-off, power test access mechanism insertion for testability.
- Examples of design decisions at the circuit abstraction: transistor sizing, metal-routing, body biasing optimisation, buffer insertion for wire delay minimisation and sizing for gate delay minimisation.
- Examples of metamodels for design decisions include Bash/Tcl/SKILL/Yaml grammar (syntax) and semantics used for design flow management, e.g. sequence of evaluation/transformations/design steps that apply, piping of intermediate results, design objects management, and visualisation and reporting.

Remark 23 (Refinements, transformations and design decisions). In some literature [18], transformations represent the primary concept to encapsulate refinements of models through two means: semantic preserving and semantic non-preserving transformations, wherein design decisions are perceived in this context as semantic non-preserving transformations. In OMG® model-driven architectures, model-to-model transformations serve as the primary methods of model manipulation, in the broadest sense. In EDA, and specifically in RT and TL, synthesis and compilation are the prevalent terms used to bridge the abstraction/refinement gaps from specifications to implementations. Within the MoD concept, the decision was made to position design decisions as the overarching concept, encompassing transformations, compilation, synthesis, satisfaction and optimisation problems. We think that 'designing' is the central activity here, leading to the choice of using the term design decision as an umbrella concept for these optimisation, model transformations, and refinement activities. This includes variable assignment algorithms, as seen in multi-objective optimisation and satisfaction problems, in addition to semantic-preserving and semantic-non-preserving endogenous (intra-language) and exogenous (inter-language) transformations.

From a categorical perspective, transformation, abstraction, and design decisions can exhibit several properties often associated with morphisms in a category. Let us discuss each of these properties:

1) Commutativity: Within category theory, commutativity often describes diagrams where morphisms can be composed in varying orders, leading to the same result. Relating to design decisions, this property suggests that certain decisions, when made in different sequences, can still culminate in the same overall system design. Nevertheless, it is crucial to recognise that not all design decisions are commutative, as the sequence of decision-making can influence the final design profoundly.
2) Distributivity: For design decisions, distributivity could mean that a combined design decision (a series of individual decisions) can be consistently distributed across various facets of the system design. For instance, a joint decision to refine both the system architecture and functional specification might be segregated into separate decisions for the architecture and the functionality.
3) Reflexivity: Reflexivity in category theory implies that every object has an identity morphism mapping it to itself. In terms of design decisions, this can be equated to a 'trivial' decision, leaving the design unaltered. Related to that is the identity morphism. This refers to a particular design decision that, when enacted, leaves the design untouched, stemming from reflexivity. It is akin to choosing to maintain the current design.
4) Nilpotent: A nilpotent design decision, when repeated or compounded with itself a certain number of times, could lead to a trivial decision, effectively 'resetting' the design.
5) Idempotent: An idempotent design decision, upon repeated application, results in no further design changes beyond its initial influence. Such decisions can set a particular design feature to a fixed state.
6) Inverse Transformation: Here, an inverse transformation would reverse the effects of an initial decision when applied subsequently, restoring the design to its previous state.
7) Products: In category theory, a product of two objects captures their 'shared information'. In the realm of design decisions, a product might signify a collective decision affecting two subsystems or design components. This 'product decision' would embody decisions made at the intersection of these components, with the morphisms (decisions) for each component detailing its respective impact.
8) Coproducts: Representing the 'sum' or 'union' of objects in a category, a coproduct decision in design could merge independent decisions concerning distinct design components. Such a decision influences each component independently, with the cumulative design impact being the 'union' of effects on individual components.
9) Limits and Colimits: In category theory, when we say a diagram 'commutes', we mean that there exist unique paths that lead to the same outcome, regardless of the route taken through the diagram. Limits and colimits are notions central to this idea. The limit captures the 'smallest' object over which a given diagram commutes. In the context of design decisions, this could be interpreted as the minimal set of decisions necessary to implement a specific design feature. Conversely, colimits can be viewed as indicating the maximal design impact achievable when a certain set of decisions is applied.
10) Monoidal Construction: A monoidal construction integrates a bifunctor (a functor of two arguments) denoting a 'tensor product' operation and a unit object serving as this operation's identity. Regarding design decisions, a monoidal category might depict the structure of amalgamating decisions (via the tensor product) and the existence of a 'do nothing' decision acting as this

operation's identity. It offers a structured way to reason about the amalgamation and sequencing of design decisions.

Definition 9. Design Rules (Λ)
Let $\mathcal{M} = \langle \text{MoS}, \text{MoA} \rangle$ be a source category where objects signify either models of specification or models of architecture, and morphisms represent transformations between these models. Let $\mathcal{I}$ be the target category with objects characterising models of implementation. *Design rules*, denoted by Λ, is posited as a functor:

$$\Lambda : \mathcal{M} \to \mathcal{I}$$

This functor maps:

- Objects in $\mathcal{M}$ to constraints in $\mathcal{I}$, refining the implementation to adhere to the stipulated rules.
- Morphisms in $\mathcal{M}$ to morphisms in $\mathcal{I}$, conveying the influence of rules on transformations from specifications and architectures into implementations.

Design rules, encoded in a coherent linguistic structure, stipulate the requisite conditions a design must satisfy to ensure its correctness. This linguistic perspective provides a framework for capturing, interpreting, and deploying rules. When coupled with design decisions Δ, these rules offer a pivotal layer of interpretation, guiding the refinement of the MoS and MoA to yield a "correct" implementation in the MoI. In specific design contexts, such rules can be collectively referred to rules deck.

From a categorical perspective, design rules can be represented in a category, $\mathcal{C}$, as follows. The objects in $\mathcal{C}$ correlate to the various elements of the design system, including specifications, architectures, evaluations, design decisions, and implementations. The morphisms in $\mathcal{C}$ denote the composition grammar or rules, defining how these elements interact and combine. This formalisation encapsulates the relationships and rules among different design components. The rules ensure the validity and compliance of the design by offering constraints for the components and their interactions.

Example 10 (Design rules). Examples of design rules include:

- *System/Transaction Space*: Spatial and temporal isolation criteria for hypervisor partitioning in mixed-critical systems.
- *RT Abstraction*: Low-power design rules, design-for-testability rules.
- *Software*: Programming guidelines such as MISRA-C:2004 for C programming language.
- *Circuit Space*: Electrical rules, design-for-manufacturability (DfM) rules, design-for-yield rules (DfY), antenna rules, rules for electrostatic discharge (ESD) and electro-migration (EM).
- *Physical Space*: IPC rules for manufactured printed circuit boards such as IPC J-STD-001, IPC-A-600, IPC-A-610, IPC-A-620, IPC-6012, IPC-7711/21, IPC-7251, IPC-7351.

Design rules often emerge as extra-functional requirements identified during design steps once a particular implementation is selected. These rules tend to be specific to platform vendors and manufacturing foundries, acting as a feasibility assessment for the design. Design rules can be equated with contracts in a contract-based design, wherein the criteria for design correctness are articulated.

Definition 10. Model of Implementation (MoI)
Let $\mathcal{I}$ be a category where objects encapsulate the various characteristics of system implementations, and morphisms depict transformations and interactions between these implementations. Within this categorical context, the *model of implementation* (MoI) can be construed as a formal modelling language, bearing similarities to both conventional programming and hardware description languages.

An object in $\mathcal{I}$ represents a parameterisable implementation, compatible with a refined model of functionality MoF′, a refined model of architecture MoA′, design rules Λ, and extra-functional specifications (MoX). Formally, this association can be expressed as:

$$\begin{aligned} &\text{MoI} : \langle \text{MoS}' \times \text{MoA}' \rangle \\ &\text{s.t. MoI} \models \text{MoA} \models \text{MoS} \\ &\quad \& \ \text{MoI} \vdash \Lambda \cup \text{MoX} \qquad\qquad \text{MoF} \times \text{MoA} : \\ &P_p \times C_p, (I_p \in \text{MoF}) \times (I_p \in \text{MoA}), \\ &(G_h \in \text{MoF}) \times (G_c \in \text{MoA}) \end{aligned}$$

Here, $\models$ and $\vdash$ should be interpreted as *models* and *satisfies* respectively.

Remark 24 (Refinement and derivation of a MoI). The term *refined* highlights the distinction between the original models and their subsequent versions in the implementation. The MoI pertains to the 'physical/geometric' domain of the Gajski-Kuhn Y-Chart or the platform-specific model (PSM) in the OMG's model-driven architecture (MDA) terminology. Generally, the technology, platforms, or target implementation libraries constitute the model of implementation.

Remark 25 (Functional-architectural mapping and implementation complexity). The design space, often referred to as the implementation space, emerges from mapping functional capabilities to architectural opportunities, guided by valid mappings (design rules and composition grammar/syntax). The complexity of this space can escalate swiftly for cross-space designs due to the exponential growth of possibilities at each abstraction level. To control this, it is imperative to restrict the options within each complexity domain to a level that facilitates optimal results.

Example 11 (Examples of MoIs). 1) With functional specifications in RT abstraction space defined using the synthesisable subset of VHDL and extra-functional

specifications like multi-mode multi-corner (MMMC) or Synopsys design constraints (SDC), and an architectural netlist of standard cells in Verilog, the MoI might be a GDSII/OpenAccess implementation supported by foundry-specific PDK and LEF files.

2) Starting with an (MoF, MoA) pair such as ((SystemC, Synchronous dataflow MoC), LSLA in AADL), the MoI might consist of C programmes, an implementation of the Synchronous dataflow MoC with limited buffers, and a shared memory bus system detailed in AMD/Xilinx microprocessor hardware specification (MHS) compatible formats. Here, MoC and MoA denote the models of computation and architecture, while MoX embodies extra-functional specifications such as latency and area footprint.

4) EVALUATION

- Evaluation within design processes can be encountered in many different stages:
  1) before design for early feasibility assessment
  2) during design for making design decisions and design space exploration
  3) after design to verify the design outputs, including verification exercises,
  4) after implementation/manufacturing to diagnose and troubleshoot problematic designs.
- Evaluation models can exist across the different abstraction spaces in the physical abstraction space where thermodynamics and electromagnetic interference are analysed in printed circuit boards and modules. In circuit abstraction space, evaluation models are used to assess transient effects, steady-state conditions, variability, and noise analysis in addition to random stochastic processes analysis. In RT abstraction space, static timing analysis and average power are predominant in digital circuits. In system space, functional safety, and performance are among the few relevant characteristics of commercial and industrial products and systems.
- Evaluation models can include the formal verification and functional simulation of functionality, e.g. equivalence checking in addition to the performance and cost evaluation or analysis given a point in the design space that belongs to the implementation. This can include formulas and theorems for real-time systems and models of computation such as consistency, liveliness, deadlock, throughput, buffer size, latency, schedulability. Technology, platforms or implementation target libraries such as DEF/LEF that are typically part of the model of implementation are also usually shared for the model of evaluation for evaluation purposes.

Definition 11. Model of Evaluation (MoE)
A *model of evaluation* is a collection of tools, frameworks, or/and methods, which may be composite, that stipulates the specific procedure and relationships required for evaluating both the functional and extra-functional properties of a model of implementation (MoI). This evaluation seeks to answer the question: 'To what extent does the MoI satisfy the model of specification (MoS)?', which can be symbolically represented as: $MoI \models MoS$? The MoE serves as a functor in the category of model design, transforming the MoI to assess its adherence to the MoS. In mathematical terms, this can be expressed as:

$$\text{MoE} : \langle \text{MoI} \rangle \rightarrow \text{MoS}$$

Furthermore, when we elaborate on the components of the extra-functional specifications (MoX), the MoE morphism can also be expressed as:

$$\text{MoE} : \langle \text{MoI} \rangle \xrightarrow[\vartheta \in \text{MoX}]{} \varrho \in \text{MoX} \cup \text{MoF}$$

In this context, ϑ represents a given element within the MoX, while ϱ signifies the resultant evaluation of the MoX, which is combined with the model of functionality (MoF) to provide a comprehensive evaluation of the MoI.

An alternative categorical counterpart for the model of evaluation (MoE) as a category can be described as follows:

Let $\mathcal{C}$ be a category that represents the model of evaluation. The objects in $\mathcal{C}$ correspond to the different implementations, denoted as MoI, at a specific level/space of abstraction. The morphisms in $\mathcal{C}$ represent the evaluation procedures and relationships that map an implementation to the corresponding specifications, denoted as MoS. This categorical description provides a framework for studying the evaluation of implementations and their correspondence to specifications. It allows for the formal analysis and verification of functional and extra-functional properties, aiding in the assessment and validation of the design.

The evaluation model in Definition III-A4 does not seem to include the case of executable specification, where one can simulate the effect of the specifications given a set of inputs to check and evaluate whether it provides the right results, i.e. validation of whether we are building the right thing? Since an implementation model at a specific level/space of abstraction can be considered to be the specification model of the next/subsequent space/level of abstraction, one can still see a notion of the term where the specification model can be evaluated using the evaluation model, by assuming that $\text{MoS}^{i+1} \simeq \text{MoI}^{i}$. Evaluation models and other decision-making models may use extended models iof architecture and specification, or augment such models to generate design outputs. For example, superimposing inputs/loads/power/electric/noise sources for system simulation and environmental stimuli to exercise the system in relation to the extra-functional analysis of interest. Additionally, design decisions and evaluation models can

employ various intermediate languages and procedures to capture different aspects of the design automation processes as required.

Example 12 (MoEs).

- Simulation models for metal oxide semiconductors field effect transistors (MOSFET) such as BSIM family [28] is an example of a *MoE* with multi-objective evaluation functions at the device/physical level.
- Simulation program with integrated circuit emphasis (SPICE), such as PSPICE, LTSPICE and Spectre is an example of a *MoE* with a multi-objective evaluation system at the circuit level.
- Instruction-set computer simulators such as Gem5, Simics and ARMulators; virtual simulation platform, such as OVPSim and Imperas OVP populated with power interception library and ARM A processor models can be an example of a *MoE* for instruction/cycle-accurate functional (and time/power extra-functional) evaluation at the transaction level.
- Hardware description language (HDL) functional simulators and static timing analysis engines like Intel QSim, and Cadence XCellium/Tempus are examples of *MoE* for the behavioural and functional specifications at the RT abstraction. Processor-based or FPGA-based *emulation* platforms such as Cadence Palladium or Protium can be an example of *MoE* at the RT for evaluation of the functional specification.
- Logical/Sequence equivalence checking (EC) and parasitic extraction are examples of *MoE* that check the equivalence of behavioural specifications in relation to transformed versions in the implementation model at the RT and logic abstractions.
- Matlab Simulink, OPNet and NoCSim are examples of *MoE* to evaluate the functional and extra-functional aspect of system design at the transaction levels.

Definition 12. Characterisation
Characterisation of evaluation model, is the process that aims to resolve unknown implementation- or technology- or platform-dependant parameters or attributes or constants.

Characterisation or Identification (or sometimes called analysis or profiling in the software domain at the system space) is usually a vendor and platform-specific exercise aims at identifying technology-dependent characteristics necessary for the evaluation of the system such as average/best/worst case communication/execution/response time. In the RT domain, FPGA vendors and foundries characterise their technology-specific devices and manufacturing process (PDK) into corresponding implementation templates and datasheets on the characteristics of the end product (electrical, mechanical, timing, power, failure-in-time, etc.).

Example 13 (Characterisation). Examples of characterisation or model identification may include:

- At the transaction level or system space: software processes memory footprint characterisation, execution time characterisation, memory access for transactions, intra- and inter-chip communication time per transaction, baseline system power consumption, system space failure-in-time characterisation.
- At the RT space: fall and rise time output propagation delay per gate and per IP, wireload delay, gate area characterisation, input and output capacitive/resistive loads for logic gates, toggling rate or switching activity probabilities for power estimation. This is apparent for IEEE 1801 and the corresponding TSV/SAIF/VCD annotation framework for power evaluation and the defacto industry standards '.lib' liberty formats for timing parameters.
- At the circuit space: threshold voltage characterisation, constant characterisation of electrical parameters of resistors, capacitance and other elemental circuit components, maximum and minimum load per circuit component, inductance/capacitive/resistive parasitics extraction (IEEE 1481).
- At the physical space: device reliability, e.g. failure in time of single event upsets, for memory devices.

5) DEVELOPMENT

The development of computer applications or systems typically occurs within the context of engineering projects. The primary aim of these projects is often to enhance existing products or technologies or to introduce new ones. As products and technologies mature, they can be described using different modelling maturity levels or technology readiness levels. For instance, within the aerospace industry and referencing NASA's technology readiness level (TRL), a product might be ascribed level 1 when a prototype is built, demonstrating its foundational principles. Conversely, when the product has been proven in an operational environment, it might attain level 9. To differentiate the design as it progresses through extensive developmental stages like TRLs, we employ the term 'epoch'. While TRL is a staple in aerospace engineering and research projects, serving as a motivating factor for the introduction of developmental stages, similar concepts can apply to other sectors. For instance, in other industries, engineering or customer change orders might reflect design modifications which can be catalogued as shifts along developmental stages.

Within each extensive developmental stage, computer systems might undergo several smaller, either automated or manual, engineering processes or steps. These steps are sometimes encapsulated by design, development, or process flows and are disseminated among members of one or more engineering teams. Given their relative brevity compared to larger developmental stages, we refer to them as 'sub-epochs'. The time discrepancy between the design before and after each of these smaller steps is also termed a sub-epoch. From this perspective, sub-epochs represent intermediary

steps in a design flow. In mature computer-aided design fields, like register-transfer level integrated circuit designs, automated design flows facilitate the transformation or refinement of high-level descriptions (e.g., RTL models) into lower-level physical formats (e.g., GDS-II). Notably, these design flows invoke various software tools to shift the design from initial specification to final implementation models. Typically, these flows are nonlinear, iterative, and encompass several intermediary phases and representations. Throughout these intermediary steps, specification models experience various phases, including architectural refinement, operational scheduling, resource allocation, functional simulation, PPA evaluation, and synthesis layout.

Another manifestation of sub-epochs is within software development methods and versioning. Agile or scaled agile frameworks (SAFe), for instance, are commonly employed in software development. Within agile or SAFe, temporal intervals, such as increments (spanning five iterations/sprints) or iterations (usually two-week periods), help structure the developmental stages of a software feature. Sub-epochs can be used to chronicle these increments and iterations. Furthermore, epochs and sub-epochs can serve to distinguish between major and minor software versions, capturing the evolution through various versions.

6) MODEL OF DESIGN

Drawing upon the aforementioned core components, namely models of specification (MoS), architecture (MoA), implementation (MoI), evaluation (MoE), and design decisions/rules (Δ/Λ), we are now in a position to present the *model of design (MoD)*. This model can be viewed as a 2-category or higher-order category that captures the essence of the design problem.

Definition 13. Model of Design (MoD)
A *model of design* (MoD) is a higher-order category that formalises the design problem for potential design specifications belonging to a specific model of specification (MoS) and architectures (MoA) within an arbitrary abstraction space. It uses design decisions/Rules(Δ/Λ), given models of evaluation (MoE), to generate an implementation under a model of implementation (MoI) at a specific abstraction space.

The MoD can be hierarchical, encompassing various components and sub-design problems, and can evolve over time to incorporate multiple development stages. MoD can also be composed with other MoDs. Formally, we define MoD as:

$$
\begin{aligned}
&{}^{\mathrm{o|m|}}\mathrm{MoD}^{\mathrm{A.s}}_{\tau.\mathrm{e}} : \langle {}^{\mathrm{o|m|}}\mathrm{MoS}^{\mathrm{A.s}}_{\tau.\mathrm{e}}, {}^{\mathrm{o|m|}}\mathrm{MoA}^{\mathrm{A.s}}_{\tau.\mathrm{e}} \rangle \\
&\xrightarrow[{}^{o|m|}\Lambda^{A.s}_{\tau.e},\, {}^{o|m|}\Delta^{A.s}_{\tau.e}]{{}^{\mathrm{o|m|}}\mathrm{MoE}^{\mathrm{A.s}}_{\tau.\mathrm{e}}} {}^{\mathrm{o|m|}}\mathrm{MoI}^{\mathrm{A.s}}_{\tau.\mathrm{e}} \\
&\text{s.t. } {}^{\mathrm{o|m|}}\mathrm{M}^{\mathrm{A.s}}_{\tau.\mathrm{e}} : \langle \bigcup_{\forall c} {}^{\mathrm{o|m|}}_{\mathrm{c|}}\mathrm{M}^{\mathrm{A.s}}_{\tau.\mathrm{e}} \rangle
\end{aligned}
$$

where A: abstraction space, s: sub abstraction space, $\tau.e$: development time epoch, m:meta-model, o:object instance of the model, c: component.

From a category theory perspective, we can define the model of design as follows: Let $\mathcal{C}$ be a 2-category (or higher-order category) that represents the model of design. The objects in $\mathcal{C}$ are categories representing different components and sub-design problems, denoted as MoS and MoA, at specific abstraction spaces and development time epochs.

These objects capture the design problem, the design specifications, and the implementation, while the arrows describe how this design is realised and evaluated. Specifically, the morphisms (or functors) in $\mathcal{C}$ represent the design decisions/rules (Δ/Λ), and the 2-morphisms (or functors) represent the evaluation models that map morphisms to other morphisms. This effectively transforms the design specifications into the implementation and evaluates whether this transformation has been carried out correctly.

Central to our discourse is how model of design (MoD) concept encompasses perspective on system design. Rooted in various foundational design methodologies and taxonomies, the MoD concept reinterprets and extends these methodologies, integrating them within a single, unified framework. Figure 1 provides a visual representation of the MoD and its intricate relationship with the Gajski-Kuhn Y-chart, among other methodologies.

The MoD concept, depicted in the figure as triangles of refinements embraces a hierarchical approach, capturing models for individual components that collectively constitute the complete design. This is articulated within the component, c: component in ${}^{o|m|}_{c|}MoD^{A.s}_{\tau.e}$, facilitating hierarchical designs across diverse abstraction spaces. Such a methodology proves invaluable in ensuring design reuse and effective encapsulation of design concerns for intricate sub-domains of hardware and software components. Examples include general-purpose processors, graphical processors, memory systems, I/O modules, hypervisors, and real-time operating systems. Guided by the MoD framework, distinct teams can systematically develop design models for architectural, evaluative, and implementational facets of various modules and components, such as application processors, graphical processors, communication processors, and clocking modules.

It is essential to position the MoD concept within the broader landscape of system design methodologies, particularly for embedded computing systems. The MoD concept aligns with established academic system design methodologies, including platform-based design (PBD), component-based design (CBD), and model-based design (MBD). In comparison to the double-roof model for hardware/software co-synthesis, the MoD concept introduces unique nuances. While the software/hardware implementation models correspond to our model of implementation (MoI), and top-level specifications are analogous to our model of specification (MoS), the MoD distinguishes itself

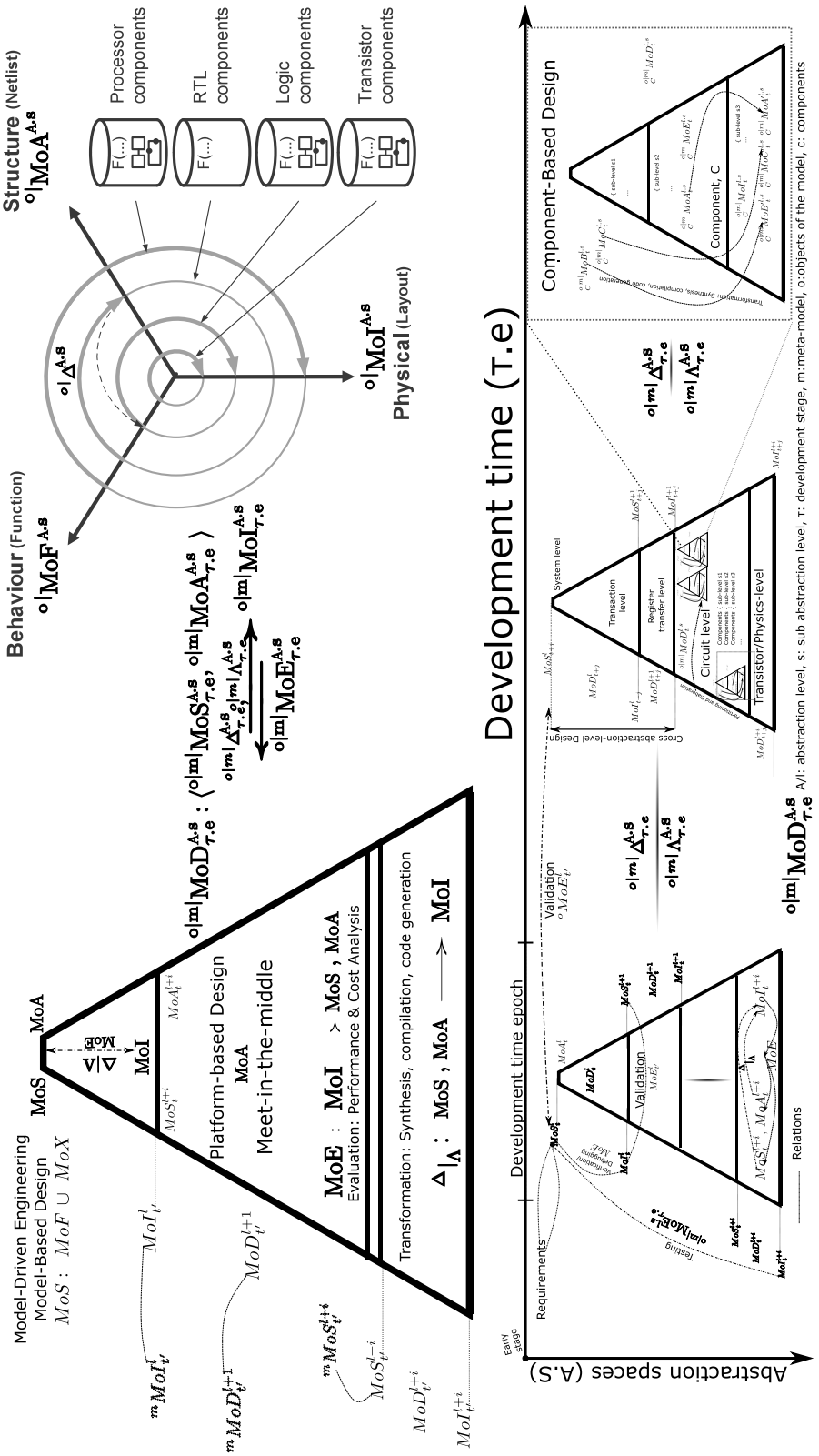

FIGURE 1. A diagram illustrating MoD concept and its main constituents: model of specifications (MoS, comprising model of functionality, MoF, and model of extra functionality, MoX), model of architecture (MoA), model of implementation (MoI), and model of evaluation (MoE), in addition to design decisions/rules across different levels of (sub) abstractions and hierarchies (see Foundational Construct 2). The diagram depicts equivalence of MoF, MoA and MoI with the behavioural, structural and physical domains of the Y-chart. It also shows the analogy of the MoD with Gajski-Kuhn Y-chart and how design decisions map to design automation activities.

through its emphasis on evaluation models and development stages.

Examining the MoD in relation to industrial practices, such as the V-chart, reveals similarities, especially concerning developmental stages like architecture, design, and development engineering. However, the MoD again differentiates itself by introducing explicit abstractions and evaluation models for design components. The seminal Gajski-Kuhn Y-chart has profoundly influenced the MoD concept, sharing many of its foundational elements. Yet, the MoD enhances this by introducing an explicit representation of extra-functional facets and evaluation models.

Delving deeper into the connections with different development frameworks, it becomes imperative to visualise the MoD framework in relation to established design methodologies and taxonomies. This comparative analysis accentuates the enhanced scope and inclusivity of MoD, particularly highlighting how it integrates diverse methodologies into its schema.

Remark 26 (On objects, models and metamodels). Models can belong to different libraries of models (e.g. specification libraries, evaluation libraries, component libraries, design libraries, implementation libraries, etc.). In each of these models, we can derive specific instances that satisfy the model we call them objects; e.g. a Sobel graph following SDF is an object or an instance while the general SDF model is a model at the transaction level. The set of all pins, ports, nets, cells, clocks of a design can be the objects of an architecture at the RT abstraction. The models can correspondingly have meta-models ($^{\mathrm{m}}$MoS, $^{\mathrm{m}}$MoA, $^{\mathrm{m}}$MoE, $^{\mathrm{m}}$MoI) with a potential addition of other languages that enables manipulating the model itself: its creation, management and modification. In various cases, meta-models can be languages to describe complementary (meta) data for logging design steps and reporting intermediate/final design results, other information such visualisation and views (e.g. graphical/symbol schematic/physical view, and simulation purposes view), or other info e.g. versioning, compatibility and executability. The concepts of object, model and meta-model within the model of design can be analogous to that from the OMG meta object facility, whereby the M0-level maps to the objects compliant with the models ^{o}MoD, the domain-specific model M1-level maps to the MoD models MoD, and the metamodel M2-level and the meta-language M3-level maps to MoD metamodels, ^{m}MoD.

Remark 27 (MoD and MoD constituents as categories). An idea behind our definitions of MoD and its constituents as models is to enable treating them as categories and defining corresponding morphisms (in the sense of category theory). This means MoD would comprise categories of specifications, architectures, and implementations (called categorical objects or the vertices on the category graphs). Evaluation models, design decisions, and rules would then be morphisms, functors and natural transformations that relate the MoD categorical objects to each other (the arrows on the category graphs). In that sense, operations on MoD and its constituents such as unification, categorical products, and transformations can be formally defined. An illustration of the MoD as a category of categories is depicted in Figure 2.

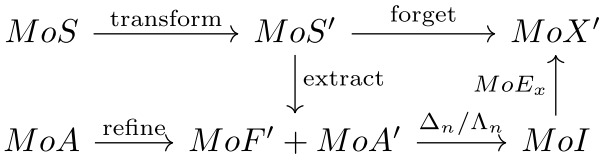

FIGURE 2. A diagram illustrating MoD as a category of categories: model of specifications (MoS, comprising model of functionality, MoF, and model of extra-functionality, MoX), model of architecture (MoA) and model of implementation (MoI). The model of evaluation (MoE) and design decisions/rules are depicted as morphisms or arrows. Note + here technically denotes a categorical co-product.

Figure 3 features a graph and range of symbols, many of which are used in category theory, to denote transformations, relations, and assumptions within and between the subsystems. MoS, Δ/Λ, MoE, MoA, and MoI represent categories in the model of design. This might stand for different aspects of a system design, such as specification (MoS), evaluation frameworks (MoE), architectures (MoA), implementation representation (MoI), and change operators or model transformation units (Δ/Λ). Id is a standard notation in category theory for identity morphisms, i.e., morphisms that leave objects unchanged when applied. The letters inside the arrows (e.g., T, I, R, A, G, C, E) represent different morphisms, functors, or natural transformations between categories, that represent various relational aspects within system design. For example, T means 'Transform', C means 'Characterise', G means 'Guarantee', etc. The symbols ($\models$, $\vdash$) near each of the nodes represent operations performed within each category or constraints applied to it. The double-headed arrows with "in" label are representing inclusion functors. These would indicate that some elements or structures from one category can be seen as a part or structures of another. The complex paths between the different categories (MoS, Δ/Λ, MoE, MoA, MoI) represent more complex transformations, perhaps involving multiple steps or the composition of multiple basic transformations.

From a systems engineering perspective, this might depict a model-based or category-based systems engineering process. Each category could correspond to a different stage or aspect of the system design process, with the transformations representing different design or analysis tasks. The diagram might express how different models and a change/relation operator or model transformation unit relate to each other and interact throughout the design's sphere. It also hints at how assumptions, restrictions, and other constraints factor into these relations and interactions.

Remark 28 (MoI circular reference in MoD). From Definition 13, MoI appears both as input to MoE and as output of MoD, which might suggest some sort of circular dependence. This is usually circumvented by provision/development of

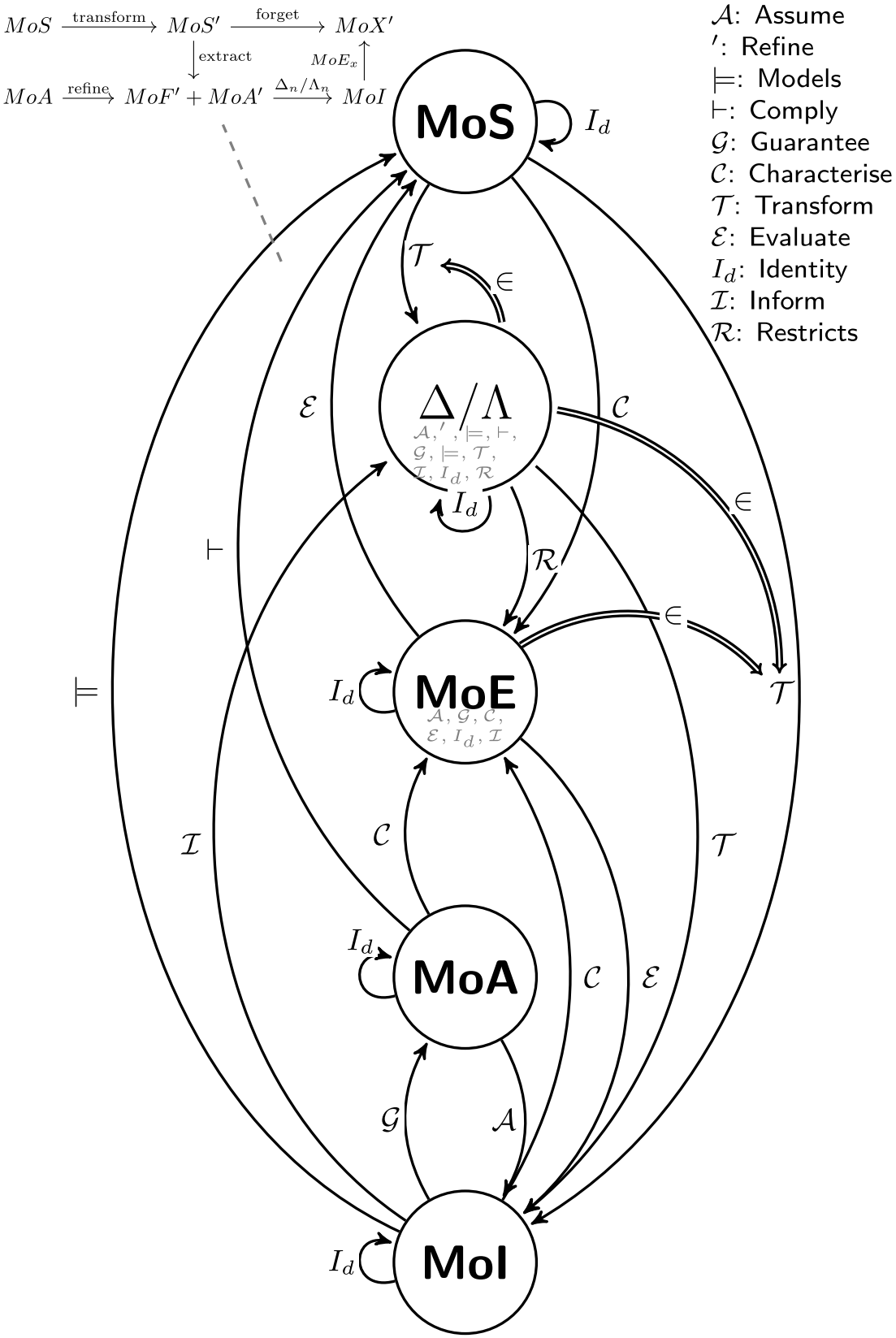

FIGURE 3. MoD as a category of categories with expanded relationships. Note that the arrows depicting different kinds of relations between the categories of MoD, which are part of the evaluation models (MoE) and/or design decisions/rules categories.

the implementation model (MoI) and how it can be evaluated (MoE) with respect to specific requirements (MoX) through characterisation (see Definition 12).

Remark 29 (Real-time systems/models and MoD). Real-time models such as periodic, sporadic, aperiodic models can be considered as union of time-related extra-functional requirement (MoX) such as deadline, latency for untimed model of computation (MoC) such as KPN on time-triggered processor based like architectures (MoA) with schedulers. Real-time analysis such as scheduleability analysis, mapping problems and end-to-end data propagation can be considered as part of evaluation models (MoE) and/or design decisions (Δ) that are used for the design problem of real-time systems. Furthermore in a way similar to our remark on MoCs hierarchy in Remark 7, using the relationship between real-time systems and model of design, one can capture different levels of hierarchies for real-time systems as shown by Stigge and Yi [29].

Remark 30 (MoD relation to platform-based design taxonomy). Platform-based design is a well-known conceptual (meta-model) framework for the electronic system level (ESL). The MoD definition, compared to a PBD taxonomy such as [30] differs in the sense that MoD framework separates the evaluation and architectural models from the platform, as there could be various ways to evaluate a platform model and that an architecture maps to different platforms thereby justifying the distinction of the notions.

Remark 31 (MoD relation to model-based methods). The role of models within the model of design concept is central and therefore several similarities can be established between model based methods such as: model driven development, model based design, model driven engineering and model driven architecture (MDA). In particular regards to the model-driven architecture methodology, a one-to-one mapping between model of specifications, model of architecture and model of implementation with computation independent model, platform specific model and technology independent model is established. In addition, the role of transformation in MDA can be directly related to design decisions in MoD.

Remark 32 (MoD relation to component-based methods). The role of components within the model of design concept is central to the model of architecture and computation (when considering processes as component) at all levels of abstraction. In circuit level abstraction, the role of components (or circuit elements) can be viewed to be larger and more significant since all computation and platforms elements are merged. The theories developed within the BIP framework (a component based construction) can be applicable to the model of architecture and the correctness of the compositionality.

Remark 33 (MoD relation to HW/SW codesign and the double-roof model). MoD is perceived to be hardware/software codesign from its inception, whereby specifications models (MoS) play the primary entry role to the design activity. In order to make the design more inclusive to hardware, the MoX concepts had been introduced explicit to the MoS, whereas to be more inclusive to software, the MoB concepts had been introduced explicitly to the MoS. The architecture is viewed as a meeting-in-the-middle for both the SW/HW aspects as opposed to the departure in the double roof model [31] whereby the implementation and architecture of hardware and software are viewed separately.

Several industry anecdotes have encouraged our adoption of hierarchical abstractions and development timelines in our formalism, taking into account all facets of the model of design concept: specifications, architecture, evaluation, and implementation. In the IEC 61508 standard, concerns regarding formal approaches underline challenges like a ''Fixed level of abstraction'' and ''Limitations in capturing all pertinent functionality at a particular stage'', thus the emphasis on developmental axes. Consequently, the design can be articulated using MoD principles, described at varying levels of abstraction based on the abstraction spaces inherent to the MoD's components. For instance, an MoD at the RT-Circuit abstraction level might have its MoC and MoF characterised as a Boolean circuit and disjunctive normal forms; MoX detailing critical path delays and input/output loads; MoI composed of standard cells detailed in LEF file formats, drawing on TSMC 2nm process design kit technology files; and MoE using inductive, capacitive, and resistive wireload alongside standard cell delay models.

The MoD concept, given its hierarchical nature, awareness of abstraction levels, partitioning, model-centric approach, and inclusiveness of platforms, aligns with or is compliant with other notions such as: model-driven engineering/design (MDE/MBD) via the specification-based framework, component-based design (CBD) through the component-based architectural framework, and platform-based design (PBD) by presuming the design process's implementation. Furthermore, as it is feasible to have multiple MoDs in principle, it is also possible to deliberate over various product lines at different development phases. As depicted in Figure, the MoD might feature 'epochs' and 'sub-epochs'. This differentiation is not strictly formal but serves to capture the progressive design processes within various developmental stages, contingent upon the industrial standards employed. For example, a firm might use engineering, product, or customer change orders/notices to demarcate different development sub-epochs, while using technology readiness levels to differentiate between various development epochs.

B. PROPERTIES

In this subsection, we delve into a comprehensive exploration of properties that follow from our model of design. These properties, integral to the core findings of our study, illuminate key notions including, but not limited to, development

time, coherence, complexity, equivalence, and correctness. Their understanding is pivotal for a holistic grasp of the broader implications and applications of our model in Subsection III-C. Let us proceed to dissect each of these properties.

Property 1. Orthogonality of Design Models (Ω)
Orthogonality of design models is a property of a model of design $\mathcal{D} \in {}^{o|m|}_{c|}MoD^{A.s}_{\tau.e}$, that describes the degree of (categorical) interactions within its constituents $\mathcal{S} \in {}^{o|m|}_{c|}MoS^{A.s}_{\tau.e}$, $\mathcal{F} \in {}^{o|m|}_{c|}MoF^{A.s}_{\tau.e}$, $\mathcal{A} \in {}^{o|m|}_{c|}MoA^{A.s}_{\tau.e}$, and $\mathcal{X} \in {}^{o|m|}_{c|}MoX^{A.s}_{\tau.e}$, $\mathcal{E} \in {}^{o|m|}_{c|}MoE^{A.s}_{\tau.e}$ are intersecting

$$\Omega(MoD) = \bigcap \mathcal{S} \in {}^{o|m|}_{c|}MoS^{A.s}_{\tau.e}, \mathcal{A} \in {}^{o|m|}_{c|}MoA^{A.s}_{\tau.e}, \mathcal{E} \in {}^{o|m|}_{c|}MoE^{A.s}_{\tau.e}$$

The property Ω, related to orthogonalisation of design concerns, is meant to measure the extent of which a model entity within a specification is repeated within the architecture and the evaluation model. Since we consider the implementation model to be the result of the design, we expect some entities from the architecture or specification to be present in the implementation. Orthogonalisation of design models can help in facilitating model exchange and reuse within different design models. Orthogonality of design models (Ω) can be derived from the provided definition of the model of design (MoD) by closely examining the relationships and interactions between its constituent components.

In the MoD, each design component has a specific role and purpose: MoS captures the problem specifications, MoA captures the architecture, and MoE captures the evaluation models. Together, they provide the necessary elements for the design. These elements are linked through morphisms represented by design decisions/rules (Δ/Λ) that transform these specifications into a concrete implementation.

The property of orthogonality (Ω), from the given definition, measures the degree of interaction (or intersection) between these elements. In highly orthogonalised design models, elements are modular and have minimal overlaps, meaning they can be modified independently without affecting the others. This is crucial for reusability and exchangeability of design models, as changes in one area will have minimal impact on others.

To compute $\Omega(MoD)$, one computes the intersection of all elements across MoS, MoA, and MoE, through the functor Ω. This gives a measure of how much these elements are intertwined. A larger intersection (higher Ω value) would imply a less orthogonal (i.e., more interdependent) design, while a smaller intersection (lower Ω value) would indicate a more orthogonal (i.e., more modular and independent) design. Therefore, in the context of MoD, Ω serves as an indicator of the design's orthogonality, helping to quantify the potential for *model exchange and reuse*.

Note 1. *It is important to note that design methods that observe the orthogonalisation of design concerns [7] can give rise to reusable design components that are potentially efficient to implement. This is because, when orthogonalising concerns, we can avoid repetition and inconsistency between the models. However, when we try to apply the orthogonalisation principle within the models themselves (intra-model level), e.g. within design decisions (Δ) and evaluation models (MoE), it could sometimes be conceptually impossible to achieve a clean orthogonalisation within design decision problems and evaluation models as it is usually the case that design decisions and evaluation overlap and give rise to inseparable trade-offs. Furthermore, orthogonalisation of concerns is not applied to the fundamental syntax/alphabet of the modelling languages, as some overlap and commonality is in fact needed between different design components to achieve interoperability.*

Property 2. Design Space (ζ)
Design Space: The design space for a model of design $\mathcal{X} \in {}^{o}_{c|}MoD^{A.s}_{\tau.e}$ represents the degree of freedom encompassing all possible permutations of computational behaviours within the functional domain and combinations of all potential implementation options, namely:

$$\zeta(\mathcal{X}) = |{}^{o}_{c|}MoA^{A.s}_{\tau.e}| \times |{}^{o}_{c|}MoF'^{A.s}_{\tau.e}|$$

Thus, **design space exploration (DSE)** is an iterative process wherein design decisions are made to assign design variables ($\in \zeta$) to one or more values from their respective value domain, after evaluation (in line with MoE and Λ), aiming to meet or optimise the design specifications ($\in$ MoS).

This definition is also intertwined with concepts such as optimisation/satisfaction or the refinement space for the design problem, delineating potential pathways for optimisation/satisfaction or refinement culminating in an implementation. The definition draws upon prior concepts: a model of design, according to Definition 13, integrates specification, architecture, evaluation, and implementation. Given that extra-functional specifications and evaluation models, as per Definitions 5 and III-A4, omit design elements and instead outline supplementary requirements or their evaluations, they are excluded from the design space. Through this exclusion process, the residue is the translation of functional specifications into architectures.

The notion of design space (ζ) and design space exploration is rooted in the model of design (MoD), quantifying the conceivable solutions that a design model might embrace. In line with the MoD definition, a model of design amalgamates all design specifications (MoS), architectures (MoA), and functional domains (MoF) pivotal to the design task. This

multidimensional expanse of viable combinations constitutes the 'design space'. Therefore, when $\zeta(\mathcal{X}) = |{}^{o}_{c|}MoA^{A.s}_{\tau.e}| \times |{}^{o}_{c|}MoF'^{A.s}_{\tau.e}|$, it fundamentally measures the aggregate number of combinations of architectures and functional domains for a specific model of design $\mathcal{X}$. In this equation, $|{}^{o}_{c|}MoA^{A.s}_{\tau.e}|$ represents potential implementations (architectures), while $|{}^{o}_{c|}MoF'^{A.s}_{\tau.e}|$ denotes functional prospects. Design space exploration thus becomes the traverse through this expanse to pinpoint the optimal solution. Here, 'optimal' hinges on the design transformation process (Δ/Λ) and the evaluation model (MoE) that assess the suitability of a particular combination vis-Ã -vis the specifications (MoS). In this paradigm, extra-functional specifications (MoX) and evaluation models (MoE) do not feed directly into the design space as they do not usher in new configurations. Consequently, the concepts of design space and design space exploration as defined herewith follow from the model of design's definition.

Note 2 (Various Kinds of Spaces)**.** *While the design space is primarily perceived as an interaction of functional and architectural space, there exist alternative interpretations of design spaces*:

1) *Design space as MoS x MoI (application-specific system design space, as in ASIC)*
2) *Design space as MoS x MoA (application-specific system architecture design space, as in FPGA)*
3) *Design space as MoX x MoA (functionally agnostic design-for-X or component-based design, e.g., RT for time, low-power design of components such as RTOS, NoCs, memories, processors, etc.)*
4) *Design space as MoF x MoA (architectural functional design, as in Kienhuis' Y-Chat)*
5) *Design space as MoF x MoI (functional-platform design space, as in PBD)*

Other associated notions describing possibilities within the models of design encompass: specification space, functional space, extra-functional space, behavioural space, architectural space, platform/implementation space, evaluation space, and design decision space.

The magnitude of the implementation space can swiftly surge to an exponential scale when contemplating every potential permutation in the design space. This burgeoning complexity holds true even when employing a platform-based design approach, where nuances exceeding the circuit and logic spheres are abstracted by the platform selection. For instance, when targeting computing systems via a platform-based design approach, there exist over 14,427 embedded computing platforms from suppliers such as mouser.com and farnell.com (excluding original equipment/design manufacturers and OS providers). Yet, in practical scenarios, this design space can be pruned due to symmetries and limitations emerging from compliance with particular design rules and MoX.

Next, we explore the Coherence property. Coherence, in a category-theoretical context, often refers to the idea that different paths through a series of morphisms (in this case, mappings or transformations from one model to another) yield the same result, essentially maintaining the consistency of the overall system.

Property 3. Coherence (β)
Coherence: A model of design $\mathcal{D} \in {}^{o|m|}_{c|}MoD^{A.s}_{\tau.e}$, is said to be coherent with respect to its constituents $\mathcal{S} \in {}^{o|m|}_{c|}MoS^{A.s}_{\tau.e}$, $\mathcal{F} \in {}^{o|m|}_{c|}MoF^{A.s}_{\tau.e}$, $\mathcal{A} \in {}^{o|m|}_{c|}MoA^{A.s}_{\tau.e}$, $\mathcal{X} \in {}^{o|m|}_{c|}MoX^{A.s}_{\tau.e}$, $\mathcal{E} \in {}^{o|m|}_{c|}MoE^{A.s}_{\tau.e}$ and $\mathcal{I} \in {}^{o|m|}_{c|}MoI^{A.s}_{\tau.e}$ iff there exists assume-guarantee relationship between specification, architecture, evaluation and implementation that are achieved by applying the design decision and rules (Δ, Λ). Or in other words:

$$\mathcal{D} \models \langle \mathcal{S}, \mathcal{E}, \mathcal{A} \rangle \overset{\Delta}{\underset{\Lambda}{\rightarrow}} \mathcal{I}\ , \forall \mathcal{S}, \mathcal{E}, \mathcal{A}, \mathcal{I}$$
$$\mathcal{I} \models \langle \mathcal{F}' \times \mathcal{A}' \rangle\ , \mathcal{I} \vdash \Lambda, \forall \mathcal{I}, \mathcal{A}, \mathcal{F}$$

The definition of coherence within a model of design expresses this principle: an MoD is said to be coherent if there is an assume-guarantee relationship between its constituents, that is, the mappings and transformations applied (through the design decisions Δ and rules Λ) produce consistent results, maintaining the integrity of the overall design.

Given the definition of an MoD, let us examine if the coherence property follows:

1) First, a coherent MoD must satisfy the assume-guarantee relationship between the specifications, evaluation, and architecture that leads to the implementation, formalised as $\mathcal{D} \models \langle \mathcal{S}, \mathcal{E}, \mathcal{A} \rangle \overset{\Delta}{\underset{\Lambda}{\rightarrow}} \mathcal{I}$. This captures the idea that for any valid set of specifications, evaluation, and architecture, applying the design decisions and rules will lead to a valid implementation, essentially saying that the process of going from specifications to implementation is coherent.
2) The second part of the definition, $\mathcal{I} \models \langle \mathcal{F}' \times \mathcal{A}' \rangle$, asserts that the implementation must satisfy the mapping from the functional domain to the architecture domain. This maintains the consistency (or coherence) of the mapping across these domains. The clause $\mathcal{I} \vdash \Lambda$ indicates that the implementation should satisfy the design rules, again asserting the consistency of the design process.

Proving coherence in this context would involve demonstrating that the design process consistently yields valid implementations that satisfy the defined rules for any given set of valid specifications, evaluation models, and architecture. This is generally a design-specific endeavor and would depend on the specifics of the design problem and the associated specifications, evaluation models, architecture, and design rules.

The use of the term 'coherence' here is more akin to its usage in software engineering or systems design, where

it refers to consistency or logical integration of various components, rather than its specific meaning in category theory.

Notably, this notion of coherence is compatible with that of Sifakus' [26] in relation to correctness-by-construction, in the sense that our criteria on coherence is subsuming Sifakis' use of transition systems, model interactions and priorities to reason about components' composability and compositionality. Additionally, one can note that this notion of coherence relates to the assume-guarantee relations in contract-based design concepts shown in [9] and [32].

Property 4. Complexity ($\mathcal{O}$)
Complexity: The complexity of design is the complexity class that indicates the space ($\mathcal{S} \in \bigcup_{\forall n_s,k_s\in\mathbb{N}}$ DSPACE $((n_s{}^{k_s}))$ $\bigcup$ NSPACE$((n_s{}^{k_s}))$ $\bigcup$ NSPACE$((2^{n_s k_s})))$ and time ($\mathcal{T} \in \bigcup_{\forall n_t,k_t\in\mathbb{N}}$ DTIME$((n_t{}^{k_t}))$ $\bigcup_{\forall k\in\mathbb{N}}$ NTIME $((n_t{}^{k_t}))$ $\bigcup$ NTIME $((2^{n_t k_t})))$ complexity in BigO notation, $\mathcal{O}(n)$ for the model of evaluation and design decisions. i.e.

$$\mathcal{O}(MoD) : (MoE \cup \Delta) \mapsto \mathcal{S}, \mathcal{T}, n_s, n_t, k_t, k_s$$

Whereby, $\mathcal{S}, \mathcal{T}$ are the space and time complexity classes respectively, while n_s, n_t, k_t, k_s are the variables describing the degree of complexity within the complexity classes. DTIME(f(n)) and NTIME(f(n)) refers to classes of problems that can be solved in a certain of degree described by f(n) in deterministic or non-deterministic time respectively. Conversely, DSPACE(f(n)) and NSPACE(f(n)) are space complexity classes containing problems that are computable by deterministic and non-deterministic Turing machines. The terms SPACE(f) and TIME(f) are used as functions that determine the space and time complexity classes of a function, f, respectively.

The complexity of a design, as defined herewith, is about how difficult it is to evaluate a design or make design decisions, and it is measured in terms of time and space complexity. Given the definition of the model of design, we can say that complexity inherently arises from the various interactions and mappings that exist within the MoD.

In the context of the MoD, the time and space complexity can be understood as follows:

- Time complexity ($\mathcal{T}$): Time complexity is a measure of the computational resources, specifically time, that an algorithm or process consumes as a function of the size of the input. In the context of the MoD, time complexity is a measure of the computational effort required to evaluate a design or make design decisions.
- Space complexity ($\mathcal{S}$): Space complexity is a measure of the amount of memory an algorithm or process uses as a function of the size of the input. In the context of the MoD, space complexity is a measure of the memory resources required to store the information about the design specifications, architectures, functional domains, and their corresponding evaluation models and design decisions.

Given these, the concept of complexity can be mapped onto the MoD as follows:

The complexity of the MoD, $\mathcal{O}(MoD)$, is a function of the complexity of the evaluation model (MoE) and the complexity of the design decisions (Δ), that is, $\mathcal{O}(MoD) : (MoE \cup \Delta) \mapsto \mathcal{S}, \mathcal{T}, n_s, n_t, k_t, k_s$.

To show that this definition follows from the MoD, we can say that for any MoD $\mathcal{X}$, the set of all possible design decisions, Δ, and all possible evaluation models, MoE, have some inherent complexity. This complexity is characterised by a time complexity class $\mathcal{T}$ and a space complexity class $\mathcal{S}$, with n_s, n_t, k_t, k_s being the variables that describe the degree of complexity within these classes.

In other words, the complexity of evaluating a design (MoE) or making a design decision (Δ) in the context of the MoD falls within some space and time complexity classes $\mathcal{S}$ and $\mathcal{T}$ respectively, thereby justifying the concept of complexity $\mathcal{O}(MoD)$.

Given a MoD $\mathcal{X}$, let $\Delta_{\mathcal{X}}$ and MoE$_{\mathcal{X}}$ be the set of all possible design decisions and evaluation models for $\mathcal{X}$ respectively. By definition of time and space complexity, there exists some n_s, k_s, n_t, k_t such that $\Delta_{\mathcal{X}}$, MoE$_{\mathcal{X}}$ $\in$ DTIME$(n_t^{k_t})\cup$ NTIME$(n_t^{k_t})\cup$ NTIME$(2^{n_t^{k_t}})$ and $\in$ DSPACE$(n_s^{k_s})\cup$ NSPACE$(n_s^{k_s})\cup$ NSPACE$(2^{n_s^{k_s}})$. Hence, $\mathcal{O}(MoD)$ is well-defined.

Therefore, we can conclude that the complexity of a design is inherent to the MoD and is determined by the complexity of the evaluation models and design decisions.

Property 5. Solvability (ν)
Solvability: the solvability of a model of design object $\mathcal{X} \in {}^{o}_{c|}MoD^{A.s}_{\tau.e}$ is a function that describes the time and space (t, s) at which the design can be solved on a given (Turing-complete) computing machine (m) with finite space and time budgets (B_t, B_s) for finding approximate (numerical) or symbolic (exact) solution for a specific input of specifications, architecture, and implementation models. When denoting a computing machine (m) of a time and space budget (B_t, B_s) with $({}^{m}\bigotimes_{B_t}^{B_s})$, we can express the solvability ν of an evaluation or design decision problem as the function that returns the time and space resources (t_r, s_r) connected with that problem i.e.

$$\nu(\mathcal{X}, {}^{m}\bigotimes_{B_t}^{B_s}) : \text{Solve}(\mathcal{X}) \xrightarrow{{}^{m}\bigotimes_{B_t}^{B_s}} s_r, t_r | s_r, t_r \in \mathbb{R}_{\geq 0}$$

$$\text{s.t. Solve}(\mathcal{X}) : f(\mathcal{X}_0, \mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_i) = \mathcal{C} \mapsto f^{-1}(\mathcal{C})$$
$$= \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i) \in \mathcal{D} | f(\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i) = \mathcal{C}$$

A: assigned values and D: Domain of values

$\mathbb{R}_{\geq 0}$ is the set of non-negative real numbers. A model is then solvable if it can be solved using time and space resources that are bounded (less than infinity) within the resources of the computing machine, i.e. $s_r \leq B_s, t_r \leq B_t$.

As most of the design decisions are in general NP-hard, since most of specific design problems are bounded, they can still be feasibly solvable in bounded time and space budget (especially on a high-performance massively-parallel computing machine). As such solvability and complexity are different, but related.

The concept of solvability is fundamentally linked to the computability and complexity theory in computer science. The ability to solve a problem is indeed a function of the resources (time and space) available and the complexity of the problem itself.

In this specific case, the solvability of a model of design is defined as a function that, given a model and a machine with certain time and space budgets, produces the time and space resources necessary to solve the design. This is a plausible definition, grounded in the understanding of computational problems.

To show that this property follows from the definition of a model of design, we must show that the design problem posed by an MoD can be mapped onto a computational problem, and that the resources required to solve this computational problem are bounded.

1) Mapping the design problem onto a computational problem: The definition of an MoD describes a process of turning a set of specifications, an architecture, and an evaluation model into an implementation using a set of design decisions. This process can be seen as a function: given a set of inputs (the specifications, architecture, etc.), it produces an output (the implementation). This function is precisely the kind of mapping we need to turn the design problem into a computational problem.
2) Bounding the resources required to solve the problem: The solvability function we have defined takes into account the resources of the computing machine, and it gives the time and space resources needed to solve the problem. If the problem is indeed solvable, these resources must be less than or equal to the machine's time and space budgets, i.e., $s_r \leq B_s$ and $t_r \leq B_t$. If the problem can be solved within these resource constraints, then it is considered solvable.

Given these arguments, the concept of solvability is consistent with the definition of a MoD and can be considered a property that naturally follows from that definition.

As pointed out, while solvability and complexity are related, they are distinct concepts: complexity refers to the inherent difficulty of a problem (often in terms of worst-case or average-case scenarios), while solvability refers to the practical ability to find a solution given particular resources. It is important to note that even if a problem is solvable (in the sense that a solution exists), it may not be feasibly solvable if the complexity is too high relative to the available resources.

The solvability can lead to another property, development time, as they are both related in terms of the time needed for the computation in the design process. The solvability property indicates the time and space resources necessary to find a solution for the design problem. This computation time is directly related to the development time, which quantifies the total time required for the design and evaluation processes within the model of design. Hence, the solvability property can be considered as one of the contributing factors to the development time.

We can incorporate this into the definition of the development time as a property of the model of design as follows:

Property 6. Development Time (ι)
The development time of a model of design object $\mathcal{Y} \in ;^{o|m|} MoD\tau.e^{A.s}$ is a function that quantifies the total time required for the design and evaluation processes within the MoD. It includes the time needed to make each design decision and the time taken for the evaluation of the design decision against the specifications, following the design rules (Λ), to convert the design specifications (MoS) into the implementation (MoI). Development Time, ι, is expressed as an integral function of all design decisions and evaluations over time, and it varies with the speed (v) and number (n) of available computational resources:

$$\iota(\mathcal{Y}, v, n) : \frac{1}{n \cdot v} \int (\Delta_{\tau.e}^{A.s} + E_{\tau.e}^{A.s}) d\tau.e$$

$\Delta_{\tau.e}^{A.s}$ represents the time consumed by each design decision at development time epoch ($\tau.e$) and sub abstraction space $A.s$. n: the number of computational resources, and v is the speed of the computational resources.

The design decisions $\Delta_{\tau.e}^{A.s}$ include the computation time required to solve the design problem, as defined by the solvability property ν.

$$\nu(\Delta_{\tau.e}^{A.s} + E_{\tau.e}^{A.s}, {}^{m}\bigotimes_{B_t}^{B_s}) :$$

$$\text{Solve}(\Delta_{\tau.e}^{A.s} + E_{\tau.e}^{A.s}) \xrightarrow{{}^{m}\bigotimes_{B_t}^{B_s}} s_r, t_r = \iota(\mathcal{Y}, v, n)$$
$$|s_r, t_r \in \mathbb{R}_{\geq 0}$$

Here, $\nu(\mathcal{Y}, {}^{m}\bigotimes_{B_t}^{B_s})$ is the solvability of the design object $\mathcal{Y}$ on a given (Turing-complete) computing machine (m) with finite space and time budgets (B_t, B_s). This property holds provided the design decisions and evaluation processes are computationally feasible within a given time constraint.

In this property, $\mathcal{Y}$ is the object instance in the model of design, $\Delta_{\tau.e}^{A.s}$ is the design decision at development time epoch $\tau.e$ and sub abstraction space $A.s$, and $\iota(\mathcal{Y}, v, n)$ is the Development time of the MoD object $\mathcal{Y}$, given a speed v and number n of computational resources. The development time

is the integral over the design decisions $\Delta_{\tau.e}^{A.s}$ with respect to the development time $\tau.e$, scaled by the speed and number of computational resources. It gives a measure of the total time required to conduct and evaluate the design within a given MoD, considering the computational resources' speed and quantity. This property is conditioned on the computational feasibility of the design decisions and evaluations within the given time constraint.

Property 7. Predictability (ρ)
Predictability: For a model of functionality or implementation $\mathcal{X} \in {}^{o}_{c|}MoF_{\tau.e}^{A.s} \cup {}^{o}_{c|}MoI_{\tau.e}^{A.s}$, given a specific set of variable assignments $\mathcal{V}$, $\mathcal{X}$ is said to be predictable iff $\mathcal{E}(\mathcal{X}, \mathcal{V})$ is a singleton with a finite value, i.e. $|\mathcal{E}(\mathcal{X}, \mathcal{V})| = 1$. A related concept, analysability (μ), is similar to predictability, except it applies to functional models.

To show that predictability follows from the model of design (MoD) as defined earlier, we need to rely on the definitions provided for the constituent models, and their relationship with the evaluation function $\mathcal{E}$ as follows:

- The definition of predictability is compatible with the definitions provided in the MoD, as it operates on $\mathcal{X}$, a model of functionality or implementation that is part of the larger model of design. This fits within the structure of the MoD, where various sub-models combine to form the overall design model.
- Predictability Condition: The predictability condition states that $\mathcal{E}(\mathcal{X}, \mathcal{V})$ must be a singleton with a finite value, given a set of variable assignments $\mathcal{V}$. In the MoD, the evaluation function $\mathcal{E}$ is used to assess design decisions. If, given the particular assignments of variables, $\mathcal{E}$ applied to $\mathcal{X}$ results in a unique and finite output, then the model is predictable.
- Analysability: The concept of analysability extends predictability to functional models, fitting into the definitions provided within the MoD, as functionality is a core aspect of the design model.

As such, predictability provides a criterion that can be used to assess the quality of design in terms of the clarity and determinacy of its outcomes given specific variable assignments. Calculating this property in a particular design problem heavily depends on the definitions and characteristics of $\mathcal{E}(\mathcal{X}, \mathcal{V})$ and certainly that of the design specifications and its transformation via design decisions. If $\mathcal{E}(\mathcal{X}, \mathcal{V})$ can be guaranteed to always produce a single, finite value given a set of variable assignments, then predictability follows naturally. However, if $\mathcal{E}(\mathcal{X}, \mathcal{V})$ can produce multiple values or is undefined for certain inputs, the predictability of $\mathcal{X}$ may not be guaranteed.

Analysability of functional models allows their unambiguous compilation and transformation to implementable models as shown in [4]. The notion of predictability is related to Kopetz' determinism in the context of distributed computing systems, and Stephan Edwards and Edward Lee definition of determinism applicable to models of computation as follows: "A physical system behaves deterministically if, given an initial state at instant t and a set of future timed inputs, then the future states and the values and times of future outputs are entailed" [33]. "Let $M = (S, I, O, C, E, B, p)$ be a model of computation (MoC) where S is the set of all legal system specifications (i.e., supplied by a designer), C be the set of all legal choices that can be made in implementing any system, I and O be the sets of inputs and outputs accounted for by the model of computation, E and B be the sets of environmental inputs and behaviours not accounted for by the model of computation, and $p : S \times C \rightarrow (I \times E \in O \times B)$ be the system implementation function for the model of computation, which takes a system specification and implementation choices and returns a system that transforms known and unaccounted-for inputs into known and unaccounted-for outputs. A model of computation M is deterministic if for all $s \in S, c \in C, i \in I$, and $e \in E$, there is some function $d : S \times I \rightarrow O$ such that $p(s, c)(i, e) = d(s, i), b$" [34], [35]. Predictability and solvability concepts can help with choosing efficient ways of solving design problems or ruling out theoretically known computations from being considered as infeasible such as Ackermann functions and haulting problems. The definition may be extended to cover stochastic processes and probabilistic distribution of values and as such allow capturing fundamentally uncertain phenomenon through the probability function $\mathcal{P}$, e.g. $0 < |\mathcal{P}(\mathcal{E}(\mathcal{X}) \in \mathcal{D})| \leq 1$

Property 8. Synthesisability (θ)
Synthesisability: for a specification model at a particular (sub) abstraction space to be synthesisable, there should exist at least a corresponding implementation model at that (sub) space or subsequent (sub) levels.

To show that the concept of Synthesisability (θ) follows from the model of design (MoD) as defined earlier, we can establish its validity by analyzing the definitions provided for the constituent models and the structure of the MoD.

- Definition Compatibility: Synthesisability is defined here in terms of the existence of a corresponding implementation model for a given specification model at a particular abstraction level or at subsequent levels. This is in line with the structure of the MoD, which considers these different abstraction levels and includes both specification models ($\mathcal{S}$) and implementation models ($\mathcal{I}$) as constituents.
- Synthesisability Condition: The condition for a specification model to be synthesisable is the existence of at least one corresponding implementation model. This is consistent with the principles of the MoD, which stipulates the transformability of specifications into implementations through design decisions (Δ) and rules (Λ). It also aligns with the assumptions that the design decisions and rules are consistent, and they lead to at least one feasible implementation for each specification.

Therefore, synthesisability as a property does follow from the model of design. Here is a sketch of the proof: Given a specification model $\mathcal{S}$, the existence of an implementation model $\mathcal{I}$ is guaranteed by the defined design decisions Δ and rules Λ. If we assume that Δ and Λ are complete (they cover all possible design decisions and rules needed to transform specifications into implementations) and consistent (they do not contradict each other), then for every specification model $\mathcal{S}$ there must exist at least one corresponding implementation model $\mathcal{I}$. Therefore, $\mathcal{S}$ is synthesisable. This proof relies on the assumption that Δ and Λ are complete and consistent. If they are not, then the synthesisability of $\mathcal{S}$ might not be guaranteed.

As opposed to model-based design, platform based design (PBD) aided by contract theory can simplify the question of synthesisability substantially due to the fact that PBD methodology starts by the assumption that there are existing models that can be used for the implementations (platforms) provided that there exists a mapping that satisfies the contractual conditions (and the specifications).

Property 9. Decidability (η)
Decidability: a model of design is said to be decidable iff it contains solvable design decisions (Δ) and evaluation models (MoE) for correctly deriving implementation, if it is synthesisiable, from specification and architectural models.

Decidability in the context of a model of design (MoD) is defined as the capacity to make solvable design decisions (Δ) and to correctly derive implementation models from specification and architectural models via evaluation models (MoE), provided that it is synthesisable.

To show that this concept follows from the MoD as defined, we can consider the following:

- Solvability: This property is a prerequisite for Decidability. The solvability of a model of design is defined as the possibility to solve the design within a certain time and space budget on a Turing-complete machine. This is a central aspect of the MoD definition, where time and space complexity are accounted for in the concept of Complexity.
- Synthesisability: As previously established, Synthesisability is a concept that follows from the MoD. It states that for a specification model to be synthesisable, there should exist a corresponding implementation model. This concept is essential for the concept of Decidability, since a model can only be decided if it can be synthesised.
- Evaluation models (MoE): The existence and functionality of evaluation models (MoE) is a part of the MoD structure. The role of these models in deriving implementations from specifications and architectural models is inherent to the MoD.

Considering these, we can claim that the property of decidability follows from the MoD. Given a model of design, let us assume that it contains solvable design decisions and evaluation models. If this model is synthesisable, then it means for every specification and architectural model, there exists a corresponding implementation model. Using the evaluation models, the implementation model can be derived correctly from the specification and architectural models. Thus, the model is decidable. This argument assumes that the design decisions are solvable and the evaluation models can correctly derive the implementation models. If these conditions are not met, then the model may not be decidable.

Note 3. *Decidability encompasses the synthesisability of the implementation and the solvability of the design model. A decision problem, characterised by a true/false outcome, is deemed decidable if a reliable method exists to ascertain the correct answer. For intricate evaluation models, particularly dealing with physical processes, decidability might be fundamentally constrained due to factors such as: unclear initial conditions and states for memory and time-invariant systems, especially those perceived as chaotic in nature; unpredictable inputs for systems of equations with feedback loops; and models that are either non-solvable or whose analysis yields ambiguous results as shown by Edward A. Lee in his works with "Determinism" [35].*

Property 10. Validity (ϕ)
Validity: for a specification $\mathcal{S}$ to be valid with respect to a model of design $\mathcal{D} \in {}^{o|m|}_{c|}MoD^{A.s}_{\tau.e}$, it should match its corresponding requirements $\mathcal{R}$. The matching is defined by a function or relation $M : \mathcal{S} \times \mathcal{R} \rightarrow \{true, false\}$, where $M(\mathcal{S}, \mathcal{R}) = true$ iff $\mathcal{S}$ satisfies all conditions imposed by $\mathcal{R}$.

To demonstrate that the validity property is upheld according to the MoD definition, we must consider the relationship between a specification and its requirements. Given the expansive definition of a model of design (MoD) as a mathematical structure encapsulating various facets of design, the connection between requirements and specifications in this scenario is not immediately clear. Nevertheless, the property definition implicitly suggests that both specifications and requirements are integral to the model. They are constituents of the MoD, perhaps portrayed as entities within its category.

Let us denote the set of all specifications as S and the set of all requirements as R. We might envisage the function M as a morphism or functor in the category representing the MoD. This function, or relationship, maps pairs of specifications and requirements to a binary truth value.

With these assumptions, the validity property can be perceived as asserting that for each specification $\mathcal{S} \in S$, a corresponding requirement $\mathcal{R} \in R$ exists such that the morphism $M(\mathcal{S}, \mathcal{R})$ returns true. Formally, given a specification $\mathcal{S} \in S$ and an associated requirement $\mathcal{R} \in R$ for which $M(\mathcal{S}, \mathcal{R})$ returns true, $\mathcal{S}$ is, by definition of validity, valid in relation to $\mathcal{R}$. Therefore, if a valid pairing of

specification and requirement $(\mathcal{S}, \mathcal{R})$ exists for every $\mathcal{S} \in S$, then the validity property applies to the full design model. This deduction relies on the presence of the morphism M and the manner in which requirements and specifications are organised and interrelated within the MoD. If the actual MoD fulfils these conditions, then the validity property naturally emerges from the MoD's definition. Otherwise, the MoD might require adjustments to cater for these elements.

Validity pertains to the correctness of the specifications model in terms of its association with the outcomes of earlier stages: be it another design output (in the context of hierarchical design), phases during design maturation, subsequent engineering change orders, or in alignment with requirement intentions (often articulated in natural language documents). The last is challenging to evaluate, but the other elements can be verified, for instance, through formal methods.

Property 11. Verifiability (υ)
Verifiability: For a design $\mathcal{D} \in {}^{o|m|}_{c|}MoD^{A.s}_{\tau.e}$ to be verifiable, there should exist a verification function $V : \mathcal{D} \times \Lambda \times \mathbb{G} \to \mathbb{R}$, where Λ is the set of design rules, and $\mathbb{G} : \bigcup G$ represents the set of all rules of compositions.

The verification function should satisfy the following conditions:

- For any design $\mathcal{D}$ and any sets of rules Λ and $\mathbb{G}$, $V(\mathcal{D}, \Lambda, \mathbb{G})$ should be a real number representing the degree of verification coverage, VC, of the design with respect to the given rules.
- V should be designed such that it quantifies the degree to which the design adheres to the functional and extra-functional properties as outlined by the rules Λ and $\mathbb{G}$.

The verification coverage VC of a design can then be defined as follows:

$$VC(\mathcal{D}) = V(\mathcal{D}, \Lambda, \mathbb{G})$$

A design is considered verifiable if $0 \leq VC(\mathcal{D}) \leq 1$.

To show the verification function is well-defined for a specific design problem, meaning it returns a unique value of VC for each set of inputs, we can examine the following:

1) Define the verification function within the MoD: As per the given property, we assume there exists a verification function $V : \mathcal{D} \times \Lambda \times \mathbb{G} \to \mathbb{R}$.
2) Show that V is well-defined: To do this, we need to show that for every design $\mathcal{D}$ and every set of rules Λ and $\mathbb{G}$, there is a unique $VC \in \mathbb{R}$ such that $V(\mathcal{D}, \Lambda, \mathbb{G}) = VC$. This will require having a method to compare the functional and extra-functional properties of a design with the rules Λ and $\mathbb{G}$, which might be dependent on the specifics of the rules and designs at hand.
3) Show that V returns a value between 0 and 1: The range of the function V is the set of real numbers, but the value returned should be a degree of verification coverage, which is bounded between 0 and 1. This might be achieved by normalising the results of the verification process, or defining the verification function in a way that it always returns a value in this range.
4) The design is considered verifiable if $0 \leq VC(\mathcal{D}) \leq 1$. This is a straightforward consequence of the previous steps, assuming V is well-defined and always returns a value between 0 and 1.

Verifiability relates to the correctness of design as a result of design decisions or evaluation. Verifiability of a design can be demonstrated through assertions, proofs based on formal methods, emulation, simulation, and virtual and physical prototypes. In register-transfer level design, logic equivalence checking (LEC), layout versus schematic (LVS) and design rule check (DRC) can be considered a typical example to contribute to the verifiability of design. Verification coverage refers to the degree to which a verification exercise or set of verification exercises addresses all specified functional requirements for a given system or component.

Property 12. Testability (ψ)
Testability: for an implementation to be testable to a degree called test coverage (TC), it must be observable and controllable to that degree with respect to testing implementation errors affecting design variables and domains.

Observability (ϖ) for a model, comprising of internal variables with value domains ($\mathcal{V} \in \mathcal{D}$), indicates the degree to which all these variables are measurable.

Controllability (κ) describes the degree to which the model variables can be changeable within its possible domain of values.

Here, the notions of observability and controllability are introduced, which are standard concepts in systems theory and engineering. Testability is then defined in terms of these concepts, along with the introduction of a degree of test coverage, which is a common measure in electronic testing and software testing. To compute the testability within a specific model of design, the following can be examined:

1) Define observability: The concept of observability in this context is introduced as a measure of how well the internal variables of a model can be measured. Formally, for a model $\mathcal{M}$ with internal variables $\mathcal{V} \in \mathcal{D}$, where $\mathcal{D}$ is the domain of possible values for these variables, observability ϖ can be a function $\varpi : \mathcal{M} \to [0, 1]$, such that $\varpi(\mathcal{M})$ measures the degree to which all variables in $\mathcal{V}$ are measurable. This can be formalised further depending on the specifics of how measurability is defined in this context.
2) Define controllability: Controllability in this context is a measure of how well the internal variables of a model can be changed within their domains. Formally, for a model $\mathcal{M}$ with internal variables $\mathcal{V} \in \mathcal{D}$, controllability κ can be a function $\kappa : \mathcal{M} \to [0, 1]$, such that $\kappa(\mathcal{M})$ measures the degree to which all variables in $\mathcal{V}$ can be changed within their domains. This can also be

formalised further depending on the specifics of how changeability is defined in this context.

3) Define testability: Testability in this context can then be defined in terms of observability and controllability. Formally, for an implementation $\mathcal{I}$, the test coverage TC could be a function $TC : \mathcal{I} \rightarrow [0, 1]$, where $TC(\mathcal{I}) = \psi(\varpi(\mathcal{I}), \kappa(\mathcal{I}))$, and ψ is a function that combines the measures of observability and controllability in some way to quantify testability.

Note that a complete proof would require more specific details about the model, the variables and their domains, and how observability and controllability are defined. Furthermore, the way in which observability and controllability are combined to quantify testability can also influence whether this property holds.

Testing and testability in this context concerns the model of the implementation and possible defects/faults that could occur in it as an unintentional result of the construction (for hardware) or development (for software and soft hardware). Test coverage is thus affected by the assumption on the fault models, the design or unit or system under test, and the observability and controllability of the system's internal components. Test coverage refers to the degree to which a test or set of tests addresses all faults presumably present in the implementable system.

Property 13. Accuracy (α)
Accuracy: for a model of evaluation $\mathcal{E} \in {}^{o}_{c|}MoE^{A.s}_{\tau.e}$ that models the true functional specification or implementation figures $\mathcal{E}_{\text{true}}$, is said to be accurate within accuracy percentage $\epsilon^{A.s}\%$ iff

$$\frac{|\mathcal{E}(\mathcal{X}) - \mathcal{E}_{\text{true}}(\mathcal{X})|}{\max\left(|\mathcal{E}(\mathcal{X})|\,, |\mathcal{E}_{\text{true}}(\mathcal{X})|\right)} \leq \epsilon^{A.s}, \forall \mathcal{X}$$

where $\mathcal{X} \in {}^{o}_{c|}MoI^{A.s}_{\tau.e} \cup^{o}_{c|} MoF^{A.s}_{\tau.e}$.

The property of accuracy can be broken down as follows:

1) The quantity $\frac{|\mathcal{E}(\mathcal{X})-\mathcal{E}_{\text{true}}(\mathcal{X})|}{\max(|\mathcal{E}(\mathcal{X})|,|\mathcal{E}_{\text{true}}(\mathcal{X})|)}$ represents the relative difference between the estimated value produced by the model of evaluation $\mathcal{E}$ and the true value $\mathcal{E}_{\text{true}}$. The denominator ensures that the difference is scaled appropriately and avoids issues with division by zero.
2) The requirement $\frac{|\mathcal{E}(\mathcal{X})-\mathcal{E}_{\text{true}}(\mathcal{X})|}{\max(|\mathcal{E}(\mathcal{X})|,|\mathcal{E}_{\text{true}}(\mathcal{X})|)} \leq \epsilon^{A.s}$ states that this relative difference must be less than or equal to a specified accuracy level $\epsilon^{A.s}$. This implies that the evaluation provided by $\mathcal{E}$ should be within the $\epsilon^{A.s}$ threshold of the true value for all models $\mathcal{X}$, where $\mathcal{X}$ is either a model of implementation or a model of functionality.
3) The condition $\forall \mathcal{X}$ indicates that the requirement holds for all models of implementation or functionality in the MoD.

Given these elements, we can see that the accuracy property aligns with the provided MoD framework.

It states that for a model of evaluation to be considered accurate, its evaluation of any model (whether it is a model of implementation or functionality) should be within a specified accuracy level of the true value. This can be seen as a measure of how well the model of evaluation is able to accurately represent the true functional specification or implementation figures for any given model.

The notion of accuracy can be useful to reason about quality of results and performance numbers reported during the various stages of developments. In practice, models used at early stage tends to be less accurate compared to those used in last stage of design or 'sign-off'. The accuracy of models can also relate to the errors made due to numerical approximation methods used to solve or optimise during decision making and evaluation methods.

On top of the aforementioned considerations, the accuracy can also be affected by 1) transformational approximation such as the reduction of real numbers in the specification space versus the standard IEEE floating formats, or 2) architectural choices such as approximate computing architectures; or 3) data compression/conversion related data losses such as those in audiovisual processing or noise-induced analogue-to-digital quantisation.

Note 4. *Valid models of specifications imply valid specifications which imply that they comply with the intended design requirement.*

Note 5. *Testable model of implementation imply the ability to apply tests on the implementation to confirm that it is free from implementation errors, faults or failures to a certain test/diagnostic coverage as a result of innate imperfection within the manufacturing process.*

Property 14. Equivalence ($\simeq$)
Equivalence between different parts of a model-of-design can be established, as far as certain essential property (or properties) is (are) concerned, if there is an isomorphism (a bijective morphism with an inverse) between the constituting models that preserves the structure. This means, for each element of a model, there exists a corresponding element in the other model, such that they produce the same output(s) for the same input(s), as far as specific concerns are in view.

Furthermore, an equivalence between models (interpreted as categories) can be defined by establishing functors in each direction between the models and demonstrating natural isomorphisms between these functors and the identity functors on each model. These functors must preserve the structure in terms of the essential properties in concern.

To investigate the definition of equivalence within the context of a model-of-design framework, especially in relation to isomorphism and category equivalences in category theory, we consider the following:

Firstly, we explore the notion of isomorphisms. Isomorphism between constituent models in MoD: For a pair of

models within MoD (let us take $\mathcal{A}$ and $\mathcal{B}$ as examples), an isomorphism between them can be defined as a pair of morphisms $f : \mathcal{A} \to \mathcal{B}$ and $g : \mathcal{B} \to \mathcal{A}$ such that $f \circ g = id_{\mathcal{B}}$ and $g \circ f = id_{\mathcal{A}}$. Here, '$\circ$' represents the composition of morphisms, and *id* is the identity morphism of a model. This definition ensures that for each element in one model, there is a corresponding element in the other, producing identical outcomes under identical conditions, thereby retaining the model structure.

Next, we turn our attention to category equivalence. The models in MoD can be interpreted as categories, with individual components serving as objects and their interrelations as morphisms. A functor $F : \mathcal{A} \to \mathcal{B}$ maps objects and morphisms in $\mathcal{A}$ to those in $\mathcal{B}$. Similarly, a second functor $G : \mathcal{B} \to \mathcal{A}$ is described. Natural isomorphisms are introduced by two sets of morphisms: $\eta : id_{\mathcal{A}} \to G \circ F$ and $\mu : F \circ G \to id_{\mathcal{B}}$, both of which meet the coherence requirements of being both natural and isomorphic. These functors and natural transformations establish an equivalence of categories, indicating the models are structurally analogous.

The equivalence property within the model-of-design framework, which borrows from both isomorphisms and category equivalences in category theory, proves invaluable when analysing various design models and scrutinising their suitability for reuse and interchange. This includes:

- *Structural analysis*: Through the equivalence property, different models can be analysed structurally. Identifying two equivalent models implies they have matching structural properties, regardless of the specifics of their individual components. This capability enables high-level comparisons of different designs or models, setting aside the finer details.
- *Model reuse*: If two models are proven equivalent under certain conditions, one model might be repurposed in the stead of the other without altering the entire system's function. This facilitates capitalising on pre-existing models, curtailing the duration and effort in crafting new designs.
- *Model exchange*: In the same vein, equivalent models can be exchanged with ease, introducing more adaptability into the design process. This is particularly beneficial in intricate systems with multiple interacting models, as analogous models can be introduced or replaced as necessary without compromising system efficiency.
- *Simplification and abstraction*: The equivalence property can be instrumental in streamlining complex models. If a convoluted model corresponds to a more straightforward one, the latter can be employed for analysis, easing the comprehension of system dynamics. This fosters enhanced abstraction, which is pivotal in navigating the intricacies of design procedures.
- *Interoperability*: Establishing equivalence between various design models augments interoperability - an essential aspect when models, shaped by disparate teams or tools, are intended to coalesce. Proving equivalence guarantees that the models can be melded without friction.

Note 6. *The equivalence property within the model-of-design framework is not only about structural congruence but also about ensuring an approximate or exact correspondence in 'essential' attributes between two models. Here, the term 'essential' is contingent upon the specific characteristics that are deemed critical or relevant to the models under examination.*

To elucidate further, equivalence in the MoD context is reminiscent of congruence, and this is particularly vivid when considering models for their interchangeability, structural resemblance, and potential reuse. To appreciate its utility, one might look towards the realm of algorithm analysis. Here, distinct sorting algorithms like bubble sort and merge sort could be seen as equivalent if the primary focus is on the end result of sorting, rather than performance metrics or their internal mechanisms.

Similarly, in design representation, diverse representations such as a visual design schema, a layout portrayal, and a textual description of the same design may be perceived as equivalent if the assessment pivots around the structural or functional integrity of the design. Though these methods of representation differ in format, they encapsulate identical fundamental design attributes. Consequently, they can be interchangeably used without compromising the core design information.

Thus, the emphasis is on discerning and juxtaposing the 'essential' facets of the models in question. By doing so, the power of the equivalence property in the MoD framework is harnessed, amplifying the analysis, reuse, exchange, and interoperability of different design models, culminating in more streamlined and potent design processes.

In essence, the equivalence property within the MoD framework is a potent tool, amplifying the analysis, reuse, exchange, and interoperability of distinct design models, culminating in more streamlined and potent design processes.

Building on the equivalence foundation and the focus on optimising design processes, we naturally progress to an equally vital aspect of the MoD framework: the intricate property of correctness. Grasping this concept, especially in its abstract form, presents a challenging yet rewarding endeavour.

Property 15. Correctness (γ)
Correctness: is a concept that is used to reason about:

- specification correctness (γ_s) implying a validated specification freedom from errors considering wrt intended requirements (relates to correctness-of-specifications),
- architectural correctness (γ_a) implying verified or equivalence checking for the architecture freedom from errors considering its correspondence to correct composition (relates to correctness-of-composition),

- evaluation correctness (γ_e) implying the evaluation freedom from errors considering the degree of accuracy associated with the evaluation wrt each functional or extra-functional metric being evaluated (relates to correctness-of-evaluation),
- design correctness (γ_d) implying that design decisions can be verified wrt freedom from errors considering the transformation from specifications to implementations honouring design rules (relates to correctness-by-design),
- implementation correctness (γ_i) implying implementation correctness freedom from errors considering faults and failures that occur during implementation e.g. during deployment of programs and manufacturing (relates to correctness-by-construction).

As such, for a design to be correct (in the sense of $\bigcup \gamma_s, \gamma_a, \gamma_e, \gamma_d, \gamma_i$), each model $\mathcal{X} \in {}^{o|m|}_{c|}MoS^{A.s}_{\tau.e}$ and $\mathcal{Y} \in {}^{o|m|}_{c|}MoI^{A.s}_{\tau.e}$ that belongs to a *coherent MoD*, $\mathcal{Y} \models \mathcal{X}$.

The concept of correctness (γ) as defined here is a comprehensive umbrella term that encompasses various dimensions of correctness related to different aspects of a model-of-design (MoD) - specifically, specification correctness (γ_s), architectural correctness (γ_a), evaluation correctness (γ_e), design correctness (γ_d), and implementation correctness (γ_i). Each of the five categories of correctness is intrinsically tied to different facets of the model. The specification, architecture, evaluation, design decisions, and implementation all have inherent correctness conditions that need to be satisfied for the model to be considered correct as a whole. The last sentence of the definition essentially states that for an implementation model ($\mathcal{Y}$) to be correct with respect to a specification model ($\mathcal{X}$), the implementation must satisfy (or model) the specifications - a necessary condition for correctness in many design and development processes. It underlines the fundamental requirement of coherence within a MoD, which necessitates congruence and consistency between all its constituent models and their transformations.

In essence, the notion of correctness underscores the coherency of a design, stipulating that the outputs of the design models should accurately mirror the provided specifications. In some design models, the verification process can be incorporated within the evaluation model itself, as expressed by the formula $MoE(MoI, MoS, \cong) = \text{true, false}$. Here, $\cong$ symbolises the categorical equivalence of models, also known as *congruence*.

The level of correctness required is directly proportional to the established criteria for acceptable degrees of accuracy, test coverage, and verification coverage. As the demand for correctness escalates, it necessitates an increase in evaluation and design efforts, thereby calling for more computational resources. This highlights a potential trade-off between the turn-around time (TAT), synonymous with the time to design, and the desired level of correctness.

The paradigm of correct-by-construction computing system design processes can be interpreted as encompassing all facets of correctness, including specification correctness, architectural correctness, evaluation correctness, design correctness, and implementation correctness. The presented concept of correctness extends Sifakus' notion by incorporating abstraction and refinement considerations between different levels of abstraction. This is particularly relevant in system or high-level synthesis, model-to-model refinement, and vertically-oriented design problems like platform-based design and model-driven engineering, exemplified in the traditional double-roof model by Keutzer et al. [7], [36].

The interpretation of correctness here diverges from the conventional correct-by-construction concept as it specifically refers to the physical production of the implementation model within the context of electronic system design, such as integrated circuits and systems-on-chips. In contrast, Sifakus' interpretation focuses on the assembly of abstract components. Further exploration of the correctness concept in relation to contracts, component-based design, interfaces, and the derivation of interesting properties and theories from these notions has been extensively addressed in Benveniste et al. [32].

C. PROPOSITIONS AND COROLLARIES

Drawing from the properties, core constituents, and foundational constructs we have outlined, this subsection presents key propositions and corollaries that underscore the significance of the model-of-design framework. Specifically, these findings illuminate the conditions under which a model of design can be automated and the criteria for ensuring its correct automation, thus offering insights into the very essence of design modelling.

Proposition 1. Criteria for Potential Automaticity
A design can be potentially automated iff it can be described as a model of design, in a way that is *decidable* and *coherent*.

Proof: The proposition states that a design can be potentially automated if and only if it can be described as a model of design in a way that is decidable and coherent. Here is how the reasoning and proof for this proposition could be laid out, based on the definitions and properties we have discussed so far:

- The essence of a model of design is that it provides a formalised, systematic way to represent and reason about design problems. It does this by capturing and integrating different aspects of the design – its specifications, architecture, and implementation – as well as the design decisions and evaluation rules that guide the transformation from specifications to implementation.

- Decidability, in this context, refers to the idea that there exists a finite procedure (algorithm) that can determine whether a given design satisfies its specifications. For a design to be decidable, it must be possible to formalise the design problem in such a way that this procedure can be applied.
- Coherence, on the other hand, refers to the consistency and completeness of the design. A coherent design is one where all the different aspects of the design - the specifications, architecture, implementation, design decisions, and evaluation rules - fit together in a consistent way and provide a complete picture of the design.

Given these properties, we can make the following argument:

1) By capturing the design problem in a MoD, we can represent it as a formal model. This makes it possible to apply systematic reasoning and computation to the design.
2) If the design is decidable, then we can construct an algorithm that determines whether a given design satisfies its specifications. This means that we can automate the process of checking the design against its specifications.
3) If the design is coherent, then all its parts fit together in a consistent and complete way. This means that we can automate the process of integrating these parts into a complete design.
4) By combining steps 2 and 3, we see that if a design is both decidable and coherent, then we can automate both the process of checking the design against its specifications and the process of integrating the parts of the design into a complete whole. This means that the design process itself can be automated.

This reasoning suggests that if a design can be represented as a decidable and coherent MoD, then it has the potential to be automated. It is important to note, though, that this does not guarantee that the design can be automated in practice – only that it has the potential to be. Practical automation would also depend on other factors, such as the availability of suitable computational resources and the complexity of the design problem.

We can give further credence to Proposition 1 as follows: by describing the design problem in model of design vocabulary means it can be translated into formal models for specifications, architectures and design rules, since all of these models are language-based (see Foundational Construct 1). The criteria on decidability and coherence imply the solvability and completeness of the model of design to use design decisions and evaluations to correctly derive an implementation. By simplifying the automation problem of the realisation of the model of design as a compilation problem that transforms a computer language to another and by process of induction from theories developed for compilers and formal languages, we can deduce that if a design problem can be captured by a model of design, it has the potential to be automated when the criteria on decidability and coherence are satisfied. □

Having established the foundational relationship between the potential for automation and the properties of the model of design, we now delve into the specific criteria that ensure correct design automation.

Proposition 2. Criteria for Correct Design Automation
A design of an embedded computing can be guaranteed to be automated and correctly implemented iff the criteria on *decidable*, *coherent*, and *deterministic* model of design are met, with:

- *valid* model of specifications, ϕ,
- *verifiable* to the maximum degree, VC = 1, and *solvable* design decisions and evaluation model, to the highest degree of accuracy (ideally absolute),
- *testable* to the maximum degree, TC = 1, model of implementation,
- ample computational resources,
- Complete consideration of practical design constraints reflected within the model of specifications, and specific design complexities, which are inherent in the design rules and architectural choices.
- a fully controlled design environment with no variability or randomness.

Proof: This proposition follows from the MoD framework as follows:

- *Validity of model of specifications (ϕ):* According to the MoD, a valid model of specifications (*MoS*) accurately represents the requirements and constraints of the design problem. If the *MoS* is not valid, the rest of the design process may not yield a correct solution, as it would be based on flawed or incomplete specifications. Therefore, validity of *MoS* is a prerequisite for correct design.
- *Verifiability and Solvability of design decisions and evaluation model:* In the MoD framework, design decisions are made based on the evaluation model (*MoE*). Verifiability ensures that the outcomes of these decisions can be checked against the *MoS*. Solvability ensures that for every design decision, there exists an acceptable solution that can be reached through the application of design rules. If *MoE* is not solvable or verifiable to a degree of 1, it may result in incorrect or unsolvable designs.
- *Testability of models of implementation:* According to the MoD, a model of implementation (*MoI*) is a product of the design process. If the *MoI* is not testable to a degree of 1, it might not be possible to fully verify that the implementation meets the specifications, leading to inability to ascertain the freedom from potential errors in the final product.

- *Ample computational resources, practical design constraints, and specific design complexities:* These factors are inherently associated with the design process. If these are not taken into consideration, the design process may not lead to a feasible and optimal solution, despite the design being decidable, coherent, deterministic, and meeting all other conditions.
- *Controlled design:* Variability or randomness in the design environment can introduce uncertainty in the design process, which can lead to incorrect or suboptimal solutions. Therefore, a controlled environment is necessary to guarantee the performance accuracy of the final design.

In the proposition, several ideal conditions are mentioned, such as maximum verifiability and testability (VC = TC = 1), highest degree of accuracy, and fully controlled environment. These ideals represent the best-case scenarios, where every aspect of the design process is under perfect control and can be measured with absolute precision. In practice, however, these ideals may not be achievable due to various limitations and uncertainties inherent in the design process.

Nevertheless, these ideals serve as guiding principles for the construction of design flows. By striving towards these ideals, one can continuously improve the design process, aiming for higher degrees of verifiability, testability, accuracy, and control. These improvements can lead to more efficient, reliable, and robust designs, thereby advancing the quality of computer system design.

To further support Proposition 2, we can consider the following: by Property 15 and the Notes 3, 4 and 5, it follows that a design process with the criteria in Proposition 2 can have implementation that are correct, to the degrees of validity, verifiability, testability and accuracy stated therewith. From the remarks and proposition 1, it follows that a design process with such properties may also be automated.

We could also employ results from category theory to strengthen the credence of the proposition through the applicability of the Yoneda Lemma to the model of design. To apply the Yoneda Lemma to the correctness of the proposition, we must translate our design problem into categorical terms and then use the lemma to deduce certain properties about our designs. The Yoneda Lemma builds on the Yoneda Embedding (see Foundational Construct 3) and essentially states: For any category $\mathcal{C}$ and an object A in $\mathcal{C}$, the functor $Hom(-, A) : \mathcal{C} \to Sets$ is representable, i.e., there exists a bijection between $Hom(X, A)$ (morphisms from X to A) and the natural transformations from $Hom(-, A)$ to any functor $F : \mathcal{C} \to Sets$. This bijection is natural in X. To apply this lemma to our model of design (MoD):

I Categorical representation of MoD: Let $\mathcal{C}$ be our category where objects are design components or modules, and morphisms represent relationships or interactions between these components.

II Applying Yoneda (lemma): Given an ideal design component A in $\mathcal{C}$, according to the Yoneda Lemma, the entire nature of A (or its specification, in design terms) can be determined by considering all possible interactions (morphisms) from all possible components (objects) to A.

Now, let us relate this to the proposition. Assume a design component A which satisfies the conditions mentioned in the proposition:

i If the model of specifications, ϕ, is valid, it means that every morphism leading into A is well-defined, ensuring the correct nature of A.

ii The verifiability and solvability of design decisions ensure that every morphism (relationship) into A from any other object (design component) in $\mathcal{C}$ is both achievable (can be constructed) and can be checked against ϕ.

iii Testability of the model of implementation ensures that the practical (or implemented) morphisms into A truly represent the ideal interactions, i.e., the morphisms in $\mathcal{C}$.

iv Using the Yoneda Lemma, we infer that if all interactions (morphisms) into A from any object are correctly defined and implemented, then the intrinsic nature (or specification) of A is as intended. Thus, A is correctly designed and implemented.

Therefore, by the Yoneda Lemma, if all criteria in the are met, the design of an embedded computing system is guaranteed to be automated and correctly implemented. This inference has transformed the problem of design correctness into a categorical one, where the Yoneda Lemma can be applied. Through this lemma, we have provided a foundational reasoning for why the proposition holds. □

From the conceptual framework of the models and the propositions given for the criteria for potential automaticity and correctness, the following corollaries and implications can be derived:

Corollary 1. Design Models Reuse
Design models within different models of design can be reused iff they belong to models that are *equivalent* ($\mathcal{X} \simeq \mathcal{Y}$, (See equivalence in Property 14).

The proliferation of different design methods, techniques, architectures, and evaluation poses a question on whether some of them can be reused within other established methodologies. By capturing established design methodologies as models of design, assessing whether or not they can be enhanced by incorporating other models (of specifications, architecture, evaluation) can be possible through Corollary 1. The corollary helps us re-frame the question to be a question on model congruence. While the corollary might seem trivial, an implication of it is that if we were to enable design results exchange between system level design, we need to establish or adopt common (standardised) languages for design capture (specification), simulation/analysis (evaluation) and implementation.

Corollary 2. Design Composability
Models within different models of design are composable iff each model is individually analysable and the superset of all the models are analysable and coherent.

As more full or partial design methods and flows become available, each with their own strengths and weakness, composing superior design flows and methods become of interest. Corollary 2 can guide such composition by imposing two criteria on the possibility of such composition on the analysability of the individual models and the coherence of the composite model. For example, to enable the construction of a composite design flow using the specification model for ForSyde [18] and the implementation model of CompSOC [37], 1) each of the models need to be individually analysable and 2) the composition needs to be semantically and syntactically coherent. This usually leads to the question on the existence of an evaluation model for CompSOC MoI that is compatible with ForSyDe MoS, and whether there exists design decisions algorithms for converting ForSyDe MoS into CompSOC MoI. This insight can also be applied to similar hardware/platform generators such as in the network-on-chip system generator (NSG) [38].

Corollary 3. Computer-aided HW/SW Codesign
General-purpose and application-specific computing systems can be described using MoD formalism. When that is made, formal methods, compilation, synthesis and optimisation methods can be used to design such systems to achieve superior quality of results with respect to extra-functional requirements or faster turn-around time with respect to design automation time, through more efficient evaluation and design exploration.

The existence of design method is not the same thing as having system-level computer-aided hardware/software co-design automated flows. In fact, as far as our literature survey work is concerned, there are no complete system-level computer-aided hardware/software co-design flow. Corollary 3 remarks that such design automation can be guided by the appropriate formalism of the design issue in question.

Corollary 4. Design Correctness
For potentially automatic and correct MoD to hold to a reasonable extent, the following principles can be considered as a consequence of proposition 2:

- complete analysable specification models, i.e. Turing-complete functional models and design-purpose complete extra-functional models, to allow effective design specification capture;
- architectures that satisfy extra-functional specifications and are predictable to allow sound design decisions;
- evaluation models, MoE, that are accurate to the extent needed by the design to allow credible qualitative results and comparative analysis.
- design decisions and evaluation models that are solvable/decidable for the design problem/objects:
- since MoD relies on coherence of constituents, a required criterion as a priori is provably correct transformational design refinements with assume-guarantee relationships between transformed components.

To have a (system level) design automation framework, does not necessarily mean it can give rise to outputs that are correct-by-design. Corollary 4, capitalises on the concept of correctness defined in Property 15 and spells out various criteria for what could be made to enable correct construction of design automation methods of computer hardware/software codesign.

Corollary 5. Constructing Design Flows
The concept of a model of design inherently accounts for development stages ($MoD^{A.s}_{\tau.e}$) and different design decisions, thus enabling us to formally construct various design flows. These flows are a description of possible design decisions at different stages ($\Delta^{A.s}_{\tau.e}$) and relevant rules ($\Lambda^{A.s}_{\tau.e}$), considering aspects such as the functional specification descriptions in MoF, extra-functional requirements in MoX, architectural solutions in MoA, evaluation frameworks in MoE, and possible implementation in MoI across different abstraction levels.

Examples for design flow construction:

- High-level synthesis Flow: We can define an MoD as a transaction-level to RTL (Register-Transfer Level) transformation process. In this flow, we start with functional specifications (MoF) and extra-functional specifications (MoX) related to latency and area constraints, which might be described in a format like SDC (Synopsis design constraints). The architecture model (MoA) could be a globally-asynchronous locally-synchronous (GALS) digital design, incorporating full-scan test logic and JTAG debugging features. For the implementation model (MoI), the target output is an RTL level HDL (hardware description language), such as VHDL. Design decisions (Δ) and evaluations might involve selecting architectures for computing and communication, control logic inference, syntactic checking, data type transformation, re-timing, and memory inference. An evaluation model (MoE) for the latency and area might include a

layout estimator and logic equivalence checks or logical functional simulations. Design rules (Λ) could cover criteria for testability and scheduleability, and rules for non-synthesisable constructs usage in System-C or VHDL. This can be further connected to various existing digital IC flows from Synopsis or Cadence, or FPGA flows from Intel, AMD/Xilinx or Microsemi.

- Reconfigurable systems flow: In this scenario, we can define an MoD as a transaction-level to combined RTL/ELF (executable and linkable format) transformation process. This flow involves partitioning of Simulink blocks into M files that can be further transformed into transaction-level for functional specifications (MoF) and extra-functional specifications (MoX) regarding latency and area constraints. Similar to the first flow, these might be presented in a format like SDC, as input for the high-level synthesis flow above. The architecture model (MoA) could be a multi-processor system-on-chip with reconfigurable features. A possible implementation model (MoI) could target devices like AMD/Xilinx Ultrascale or Intel Agilex, which have embedded local memory and ARM-A profile RISC multi-core processors for software computation described in ELF (executable and linkable format) files. This is in addition to hardware accelerator logic derived from an RT abstraction space language such as VHDL. Design decisions (Δ) and evaluations might involve architecture selection for hardware/software co-designing, software compilation, resource allocation, on-chip network design, and memory and peripheral management. An evaluation model (MoE) for the latency and area might include a memory consumption evaluator and programmable logic estimator. Design rules (Λ) could cover criteria for software debugging, hardware/software composability rules and security checks for off-chip memory and networks, and bitstream booting.

A consequence of MoD being a composite of other models, is that classes based on the complexities of the sub-models can be constructed. In the same way MoCs are specified as dynamic versus static, and architectures can be classified on basis of topologies or processor/software complexities, and evaluation models can be different depending on what analysis methods/ accuracies can give rise to, design problems can be categorised into classes.

Corollary 6. MoD as a Taxonomy
Design problems and methods can be classified under MoD formalism to enable comparative analysis between different methods and the composition of more sophisticated design methods.

An interesting implication of capturing design methods as models of design is that, it can enable posing questions that can help in design method resuse. For example, if we take two design methods Daedalus [39] and HOPES [40], and wish to use the simulation and synthesis engines provided by Daedalus for HOPES input specifications in a flow that is potentially automatic and correct, we can use proposition 2 to guide us in that exercise in examining:

1) the coherence in term of syntax and semantics of the unified model of design
2) the decidablity of the unified model of design
3) the verifiabilty of the design decisions within the context of the unified of design
4) the accuracy of the evaluation models within the context of the unified of design
5) the synthesisablity and testability of the implementation models within the context of the unified of design
6) the validity of intermediate specification models across abstraction levels, hierarchies and development stages

Another interesting implication of capturing design methods as design models is that it can enable elevated discussions regarding how a design methodology can be improved to cover different design problems, by extending the specification model or the architectural models. Other improvement works can be in relation to the accuracy and efficiency of design decisions and evaluation models.

IV. DISCUSSIONS AND RELATED WORK

In this paper we presented a model of design, which we can consider as a category of design models $\mathcal{C}$ that captures the different components or aspects of the design, such as models of specification (MoS), models of architecture (MoA), models of evaluation (MoE), models of implementation (MoI), and design decisions/rules (Δ/Λ). The objects in $\mathcal{C}$ represent the specific instances or representations of these components, while the morphisms represent the transformations or mappings between these objects. We then use the theorems and axioms from category theory, to assist reasoning about the design models such as:

1) Composition: The composition of morphisms in the category allows us to combine and sequence transformations between models. Given two morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$, their composition $g \circ f : A \rightarrow C$ represents the transformation obtained by applying f followed by g. This property ensures that transformations between models can be composed in a consistent manner.
2) Associativity: The composition of morphisms is associative, meaning that for any three morphisms $f : A \rightarrow B$, $g : B \rightarrow C$, and $h : C \rightarrow D$, the composition is associative as $(h \circ g) \circ f = h \circ (g \circ f)$. This property ensures that the order in which transformations are applied does not affect the final result. In the context of model of design, it ensures that the sequence in which design transformations are applied does not impact the overall design outcome.
3) Identity morphisms: For every object A in the category, there exists an identity morphism $id_A : A \rightarrow A$ that serves as the neutral element with respect

to composition. The identity morphism preserves the structure or properties of the object when composed with other morphisms. In the context of model of design, identity morphisms allow for the preservation of the original properties and structure of the models during transformations.

By leveraging these properties, we can reason about the compatibility between models and ensure that the transformations applied to the models preserve their properties and structure. These properties also facilitate the development of reasoning techniques and tools for analysing the impact of transformations on the design, identifying compatibility issues, and ensuring the correctness and integrity of the design process.

When examining design problems within engineering contexts, transitioning from abstract concepts to specific applications uncovers shortcomings in current design methods. These limitations include inhibiting correct automation and restricting design reuse across methodologies, often due to inconsistencies in design decisions and discrepancies in the syntax and semantics of modelling languages. In addition, gaps exist in the transformation process of several extra-functional industrial requirements and standards, thus further obstructing automation.

We observe echoes of these issues in the works of numerous researchers. For instance, in "System Design Automation: Challenges and Limitations" Joseph Sifakis [10] promotes system design as a process involving end-to-end, correct-by-construction, and scalable transformations. He advocates for achieving semantic coherency using a unified component framework and leveraging existing 'constructivity' results for the development of rigorous system design flows.

Alberto Sangiovanni-Vincentelli et al. also shared relevant insights, stating that the main challenges in adopting platform-based design methodologies (PBD) relate to the absence of precise definitions and characterisations of platforms and their associated design flows in the industry today [30], [41], [42]. This vagueness creates difficulties when transitioning designers from traditional methodologies such as ASIC flow to the PBD paradigm and developing the necessary tools to support this paradigm.

Since these challenges were first published between 2000 and 2015, no common framework resembling the work detailed here has emerged. A possible exception is B. Baily, G. Martin, and T. Andersson's 2005 book "Taxonomies for the Development and Verification of Digital Systems" (TDVDS) [43], which provided rather unifying definitions for several industrial concepts, including system models, architectures and design processes. In their work, they compiled several views on the relevant axes using four main areas: 1) temporal detail, 2) data value detail, 3) functional detail, and 4) structural detail. The work defined concise definitions for the degrees and granularities involved in each of the four dimensions. Furthermore, the taxonomy clarified the relation between the taxonomy and other existing concepts such as platform based design, hardware-software codesign, and abstraction layers in software. For example, within platform-based models, they used: functionality (model of functionality), market (guiding specification), and structure (architecture). For hardware-software co-design, they also discussed the plane describing the interactions between hardware, software, hardware-dependent software (firmware), and manufacture (implementation of system). Within the software domain, they described the relationship between high-level objects, (low-level) code, real-time system and hardware. Our model of design concept shares several of these aspects while giving emphasis for the evaluation models and design rules that were not explicitly captured in the TDVDS works.

However, other works have touched upon related concepts. Ecker and Schreiner [44], for instance, introduced the notions of model-of-design and model-of-thing, mostly discussing templates and views useful for hardware generators. While hardware generators are crucial, we argue that limiting design problems to hardware generators provides a narrow view of the concept within system-level and cross-level computer systems design or in hardware/software codesign and cosynthesis context.

Densmore et al. [30] offered a taxonomy related to platform-based design at the system level and linked this taxonomy to the Gajski-Kuhn Y-chart. Although their platform-based taxonomy is fascinating for mapping problems, it does not necessarily delve into the role of generic architectures and their relation to extra-functional specifications in the context of design.

Moreover, the RASSP taxonomy and the Rugby model by Jantsch et al. [45], [46] warrant interest. Rapid-prototyping of Application Specific Signal Processors (RASSP) workgroup published a flat taxonomy that shares many concepts with our framework, using information and time as main axes of design abstractions. However, their work lacked a hierarchical structure through architectures, evaluation, computation, and design decisions that our model of design provides.

On a different note, the Rugby model represents electronic system design as a progression from design idea (high abstraction) to physical system (low abstraction), tracked over a development time axis. They captured data and time as considerations for the design's development through abstraction levels. Yet, they overlooked the relation of design activities to the evaluation or architectural or extra-functional aspects.

Lastly, Ecker et al.'s 1996 work [47] proposed a specific model for VHDL design flow representation, adding testing to the Y-chart. While this model contributes a testing element to the design problem, it does not address verification aspects of design (i.e., are we building the thing right?) or the validation of requirements (i.e., are we building the right thing?). Also, their design cube did not clearly define the roles of architectures or the evaluation of extra-functional requirements.

As previously discussed, we consider the MoD concept to be not only compatible but also dependent on formalism like MoCs and MoA, in addition to already existing views on academic system design methodologies like: PBD, CBD, MBD, etc. When compared with design methodologies double-roof model for hardware/software co-synthesis [31], [36], we note that the software/hardware implementation models map to MoI and the top-level specifications map to our MoS; the difference between MoD concepts and the double-roof and the X-chart is the explicit distinction/emphasis on the evaluation models and the development stages. When compared with industrial practices such as the V-chart, we note the similarities of having development stages including requirement, architecture, design and development engineering efforts in the models, but we also note that MoD adds the explicit invocation of the different abstraction levels and evaluation models over design components. The Gajski-Kuhn Y-chart [15], [17] has inspired the MoD concept and therefore share all its constituents, however, MoD adds the explicit modelling of extra-functional aspects and evaluation models. OMG model-driven architecture (MDA) principles had also inspired the development of MoD concept, especially in the definition of model-to-model transformations as a critical component in design decisions, MoD adds to it the explicit invocation of behavioural models and model of computations to allow sound analytical formal transformations that are potentially correct-by-design. Kienhuis' Y-chart has significantly influenced the definition of MoD concept, but we included the development stages to allow the cross-compatibility with engineering practices, e.g. technology readiness levels and engineering change orders. Moreover, MoD extends frameworks such as Component based Design/Integration/Construction (CBD/CbC) such as BIP by the inclusion of evaluation models, design decisions, and design rules. MoD extends Platform-based Design and meet-in-the-middle concepts by the explicit inclusion of development stages. MoD extends Model based/driven Engineering/Development/Architecture (MBD/MBE/MDA) by introducing platform related abstraction layers and implementation dependent evaluation models. All in all, the formalism that can be adopted from the MoD concept can be used to allow formal methods and analyses to apply such as in [22] and [48], which in turn can create opportunities to shortening development time and reduced verification and testing costs, by virtue of correct automatic design in addition to optimal design outputs, by virtue of computer-aided optimisation.

This high-level overview leads to a few key insights into potential applications of our model of design (MoD) concept:

- The MoD framework can encapsulate various stages of design processes, useful for tracking technological readiness, specification versioning, and engineering changes (Corollary 5). It allows comprehensive capture of related design aspects like product lifecycle stages and tool versions, facilitating different development practices.
- The MoD concept can be applied beyond embedded computing systems to a wide spectrum of computing technologies including general-purpose, high-performance, and consumer systems (Corollary 3). Multiple models of functionality (MoF) and models of extra-functional specifications (MoX) can characterise a broad range of benchmarks. This method is particularly beneficial for manufacturers and suppliers of processors and electronics and is aligned with established methodologies.
- The MoD can address design challenges of emerging technologies and applications, including non-Von Neumann and more-than-Moore systems, quantum computing, and large-scale language models (Corollaries 1, 2, 3, 4). The MoD provides a structured approach to solving the solvability, automation, and correctness of such design problems, bridging the productivity gap in complex system design.
- The MoD allows for the construction of formal design methodologies by outlining design decisions at various stages and relevant rules (Corollary 5). Examples include high-level synthesis flows and reconfigurable system flows.
- With MoD, design problems and methods can be systematically classified, enabling comparative analysis and composition of sophisticated design methods (Corollary 6). This aids in identifying missing elements in existing designs and requirements for automation and CAD tools.

V. CONCLUDING REMARKS

This paper offered a *model of design (MoD)* concept for describing design problems of computing systems in a consolidated way while keeping in view the intriguing sub-problems of evaluation, model transformation and optimisation to satisfy design specifications including extra-functional ones. The key to our work is the necessity of identifying fundamental commonalities across computer design models, spanning both hardware and software applications. By discerning shared characteristics, we leveraged foundational principles from formal languages and category theory, capturing these commonalities in a structured manner that honours the inherent complexities, facilitating a unified reasoning system. We then exploit the emergent properties of our conceptual framework to derive high-level insights into diverse design challenges. To that end, we defined five main constituents: the *model of specifications (MoS)*, *model of architecture (MoA)*, *model of evaluation (MoE)*, *model of implementation (MoI)*, and design decisions/rules (Δ/Λ). The model-of-design constituents can be expressed at multiple levels of abstractions and different development stages. When using tools and models for constructing design methodologies or flows, the model of design can help to identify:

1) coherency issues between the specifications, implementation and architectural models,
2) the degree to which correctness can be achieved with regards to the accuracy of evaluation models and design decisions, verification coverage, and test coverage,
3) the efficiency of the overall system design with respect to the degree to which the evaluation models and design decisions are solvable and decidable, and
4) development strategy for the design problem across hierarchies and abstraction levels that evolves over time.

We have identified several future research avenues related to the model of design concept, including:

- Formulating theorems that facilitate the construction and reuse of design methods, catering to different abstraction levels and development stages.
- Harnessing the ontological attributes of the design model to systematically examine and understand contemporary advancements in the field.
- Establishing distinct classes within the model of design to address a range of design problems more effectively.

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their valuable feedback on the early drafts of this paper.

REFERENCES

[1] Cisco Systems. (2023). *Cisco Annual Internet Report (2018–2023)*. [Online]. Available: https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html
[2] F. Faggin, "The MCS-4: An LSI microcomputer system," in *Proc. IEEE Region Six Conf.*, 1972.
[3] S. Mazor, "Moore's law, microcomputer, and me," *IEEE Solid-State Circuits Mag.*, vol. 1, no. 1, pp. 29–38, Winter 2009.
[4] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 17, no. 12, pp. 1217–1229, Dec. 1998.
[5] A. Sangiovanni-Vincentelli, S. K. Shukla, J. Sztipanovits, G. Yang, and D. A. Mathaikutty, "Metamodeling: An emerging representation paradigm for system-level design," *IEEE Design Test Comput.*, vol. 26, no. 3, pp. 54–69, May 2009.
[6] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design Test Comput.*, vol. 18, no. 6, pp. 23–33, Nov./Dec. 2001.
[7] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.
[8] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Component-based design for the future," in *Proc. Design, Autom. Test Eur.*, 2011, pp. 1–5.
[9] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *Eur. J. Control*, vol. 18, no. 3, pp. 217–238, Jan. 2012.
[10] J. Sifakis, "System design automation: Challenges and limitations," *Proc. IEEE*, vol. 103, no. 11, pp. 2093–2103, Nov. 2015.
[11] S. Eilenberg and S. MacLane, "General theory of natural equivalences," *Trans. Amer. Math. Soc.*, vol. 58, no. 2, pp. 231–294, 1945.
[12] N. Chomsky, "Three models for the description of language," *IEEE Trans. Inf. Theory*, vol. IT-2, no. 3, pp. 113–124, Sep. 1956.
[13] N. Chomsky, "On certain formal properties of grammars," *Inf. Control*, vol. 2, no. 2, pp. 137–167, Jun. 1959.
[14] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of 'semantics?'" *Computer*, vol. 37, no. 10, pp. 64–72, Oct. 2004.
[15] D. D. Gajski and R. H. Kuhn, "New VLSI tools," *Computer*, vol. 16, no. 12, pp. 11–14, 1983.
[16] A. Gerstlauer and D. D. Gajski, "System-level abstraction semantics," in *Proc. 15th Int. Symp. Syst. Synth.*, 2002, pp. 231–236.
[17] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.
[18] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 1, pp. 17–32, Jan. 2004.
[19] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2014, pp. 372–383.
[20] A. Bouakaz, P. Fradet, and A. Girault, "A survey of parametric dataflow models of computation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 22, no. 2, pp. 1–25, Apr. 2017.
[21] S. Stuijk, M. Geilen, B. Theelen, and T. Basten, "Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications," in *Proc. Int. Conf. Embedded Comput. Syst., Architectures, Modeling Simulation*, Jul. 2011, pp. 404–411.
[22] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. 2002.
[23] T. Villa, A. Petrenko, N. Yevtushenko, A. Mishchenko, and R. Brayton, "Component-based design by solving language equations," *Proc. IEEE*, vol. 103, no. 11, pp. 2152–2167, Nov. 2015.
[24] *Interface Definition Language Version 4.2, OMG Document Number*, Object Manag. Group (OMG), Needham, MA, USA, 2018.
[25] D. A. Lamb, "IDL: Sharing intermediate representations," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 297–318, Jul. 1987.
[26] J. Sifakis, "A framework for component-based construction," in *Proc. 3rd IEEE Int. Conf. Softw. Eng. Formal Methods (SEFM)*, 2005, pp. 293–299.
[27] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.
[28] Y. Cheng and C. Hu, *MOSFET Modeling & BSIM3 User's Guide*. Springer, 1999.
[29] M. Stigge and W. Yi, "Graph-based models for real-time workload: A survey," *Real-Time Syst.*, vol. 51, no. 5, pp. 602–636, Sep. 2015.
[30] D. Densmore and R. Passerone, "A platform-based taxonomy for ESL design," *IEEE Design Test Comput.*, vol. 23, no. 5, pp. 359–374, May 2006.
[31] J. Teich, "Embedded system synthesis and optimization," Tech. Rep., 2000.
[32] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for systems design: Theory," Res. Rep. RR-8147, 2015.
[33] H. Kopetz, *Real-Time Systems Series: Design Principles for Distributed Embedded Applications*, 2nd ed., J. Stankovic, Ed. Springer, 2011.
[34] M. Lohstroh, P. Derler, and M. Sirjani, Eds., *Principles of Modeling*, vol. 10760. Springer, 2018.
[35] E. A. Lee, "Determinism," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5, pp. 1–34, May 2021, doi: 10.1145/3453652.
[36] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.
[37] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Design Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–24, Jan. 2009.
[38] J. Öberg and F. Robino, "A NoC system generator for the sea-of-cores era," in *Proc. 8th FPGAWorld Conf.* New York, NY, USA: Association for Computing Machinery, Sep. 2011, pp. 1–6, doi: 10.1145/2157871.2157875.
[39] T. Stefanov, H. Nikolov, L. Bogdanov, and A. Popov, "DAEDALUS framework for high-level synthesis: Past, present and future," in *Proc. 25th Int. Conf. Electron.*, Jun. 2021, pp. 1–6.
[40] S. Ha and H. Jung, "HOPES: Programming platform approach for embedded systems design," in *Handbook of Hardware/Software Codesign*. Dordrecht, The Netherlands: Springer, 2017, pp. 951–981.
[41] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proc. 41st Annu. Design Autom. Conf.*, Jun. 2004, pp. 409–414.

[42] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proc. IEEE*, vol. 95, no. 3, pp. 467–506, Mar. 2007.
[43] B. Bailey, G. Martin, and T. Anderson, *Taxonomies for the Development and Verification of Digital Systems*. Springer, 2005.
[44] W. Ecker and J. Schreiner, "Introducing model-of-things (MoT) and model-of-design (MoD) for simpler and more efficient hardware generators," in *Proc. IFIP/IEEE Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Sep. 2016, pp. 1–6.
[45] C. Hein, T. Carpenter, A. Gadient, R. Harr, P. Kalutkiewicz, and V. Madisetti, "RASSP VHDL modeling terminology and taxonomy," RASSP Taxonomy Working Group (RTWG), Tech. Rep., 1996.
[46] A. Jantsch, S. Kumar, and A. Hemani, "The Rugby model: A conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 1999, pp. 256–262.
[47] W. Ecker, M. Hofmeister, and S. März-Rössel, "The design cube: A model for VHDL designflow representation and its application," in *High-Level System Modeling*. Springer, 1996, pp. 83–128.
[48] I. Konnov, J. Kukovec, and T.-H. Tran, "TLA+ model checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–30, Oct. 2019.

TAGE MOHAMMADAT (Senior Member, IEEE) was born in Abu Dhabi, United Arab Emirates, in 1988. He received the bachelor's degree (Hons.) in electrical and electronic engineering from Universiti Teknologi Petronas, Perak, Malaysia, in 2010, and the master's degree in system-on-chip (SoC) design from the KTH Royal Institute of Technology, Stockholm, Sweden, in 2017.

He has accumulated over 15 years of professional experience, from 2008 to 2023, spanning control system engineering to embedded computing system development. He has contributed with his expertise to leading engineering organizations, including Cadence Design Systems, Petronas, Schlumberger, ABB, and Hitachi, as well as several governmental agencies. His industrial engineering projects encompass a wide range of technologies, such as smart single-phase ac electronic energy meters, low-power nanometric VLSI-based MPSoC testing, digital signal processor (DSP) control algorithms for high-voltage direct current (HVDC) systems, NoC-based computing platform architectural design using FPGAs, Linux-based device driver and firmware development for edge-compute graphics processing, real-time operating system (RTOS) kernel development for 8/16/32-bit microcontroller-based educational devices, and test automation of .NET applications. Most recently, he has held key roles in research and development, consultancy, and entrepreneurship with Embedded Computing Systems, Cleeven, and Topgolf, all based in Stockholm, Sweden. He is an active contributor to diverse embedded computing system design and development projects. From 2010 to 2011 and from 2016 to 2021, he was involved in research projects at both Universiti Teknologi Petronas and the KTH Royal Institute of Technology. He participated in five European Union (EU) and Swedish national research projects, focusing on the research and development of dynamic low-power embedded systems suitable for open and changing environments. Furthermore, he has taught several master's level courses on embedded systems at both institutions. His research interests include embedded hardware and software co-development, architectural system-level design, run-time reconfigurability for radiation-hardened low-Earth orbit satellite systems, nanometric-scale resistive faults modeling, and hydrocarbon micro-tremor analysis using low-frequency MEMS-based accelerometers.

Mr. Mohammadat is a Professional Member of the Association for Computing Machinery (ACM). Throughout his career, he was a recipient of multiple leadership and excellence awards, such as the Swedish Institute Scholarship and the PETRONAS Scholarship. He ranked among the top 50 in the Sudanese Secondary School Examination, out of over 100,000 participants. He has held notable elected positions in non-governmental organizations, such as an Industrial Relations Officer, the Executive Committee of the Institute of Electrical and Electronics Engineers (IEEE), Sweden, and the IEEE Region 8-Europe, Middle East, and Africa, the Chairperson of the Doctoral Committee, Sweden's United Student Unions (SFS), Sweden, the Presidium and a Board Member of the Ph.D. Chapter and the Institute of Technology's Student Union (Dr/THS), Sweden, and a Board Member of the KTH Section ST Trade Union for Civil Servants, Sweden. He is a committed advocate for educational quality and equal opportunity, particularly in the Stockholm region of Sweden. His community service activities have included being a member of the university faculty council, employment boards, and docent committees. He has served as a technical reviewer for IEEE conferences and symposia, as well as MDPI journals.