INVITED PAPER

A Platform-Based Design Methodology With Contracts and Related Tools for the Design of Cyber-Physical Systems

A platform-based design methodology that uses contracts can address the complexity and heterogeneity of cyber-physical systems by providing formal support to the entire system-design flow in a hierarchical and compositional way. The foundations of such flow and the tools supporting its deployment are presented in this paper.

By Pierluigi Nuzzo, *Student Member IEEE*, Alberto L. Sangiovanni-Vincentelli, *Fellow IEEE*, Davide Bresolin, Luca Geretti, and Tiziano Villa

ABSTRACT | We introduce a platform-based design methodology that uses contracts to specify and abstract the components of a cyber-physical system (CPS), and provide formal support to the entire CPS design flow. The design is carried out as a sequence of refinement steps from a high-level specification to an implementation built out of a library of components at the lower level. We review formalisms and tools that can be used to specify, analyze, or synthesize the design at different levels of abstraction. For each level, we highlight how the contract operations can be concretely computed as well as the research challenges that should be faced to fully implement them. We illustrate our approach on the design of embedded controllers for aircraft electric power distribution systems.

KEYWORDS | Cyber-physical systems; contract-based design; horizontal and vertical contracts; platform-based design methodology; system design automation

Manuscript received February 4, 2015; revised June 13, 2015; accepted June 29, 2015. Date of publication September 15, 2015; date of current version October 26, 2015. This work was supported in part by IBM and UTC via the iCyPhy consortium, by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by the EU Commission under the EU project FP7-ICT-223844 CON4COORD.

P. Nuzzo and **A. L. Sangiovanni-Vincentelli** are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720 USA (e-mail: nuzzo@eecs.berkeley.edu; alberto@eecs.berkeley.edu).
D. Bresolin is with the Department of Computer Science and Engineering, University of Bologna, 40126 Bologna, Italy (e-mail: davide.bresolin@unibo.it).
L. Geretti is with the Department of Electrical Engineering (DIEGM), University of Udine, 33100 Udine, Italy (e-mail: luca.geretti@univr.it).
T. Villa is with the Department of Computer Science, University of Verona, 37134 Verona, Italy (e-mail: tiziano.villa@univr.it).

Digital Object Identifier: 10.1109/JPROC.2015.2453253

I. INTRODUCTION

A large number of new IT applications are emerging, which go beyond the traditional boundaries between computation, communication, and control. The majority of these applications, such as "smart" buildings, "smart" traffic, "smart" grids, "smart" cities, cyber security, and healthcare wearables, build on *distributed, networked sense-and-control platforms*, characterized by the tight integration of "cyber" aspects (computing and networking) with "physical" ones (e.g., mechanical, electrical, and chemical processes). In these *cyber-physical systems* (CPSs) [1]–[3], computational devices monitor and control the physical processes, usually with feedback loops where physics affects computation and vice versa.

Intelligent systems that gather, process, and apply information are changing the way entire industries operate and have the potential to radically influence how we deal with a broad range of crucial societal problems. As embedded digital electronics becomes pervasive and cost-effective, co-design of both the cyber and the physical portions of these systems shows promise of making the holistic system more capable and efficient. However, CPS complexity and heterogeneity, originating from combining what in the past have been separate worlds, tend to substantially increase the design and verification challenges.

A serious obstacle to the efficient realization of CPSs is the inability to rigorously model the interactions among heterogeneous components and between the physical and the cyber sides. CPS design entails the convergence of several subdisciplines, and tends to stress all existing modeling languages and frameworks, which are hardly interoperable today. While in computer science logic is emphasized rather than dynamics, and processes follow a sequential semantics, physical processes are generally represented using continuous-time dynamical models, often expressed as differential equations, which are acausal, concurrent models. It is therefore difficult to accurately capture the interactions between these two worlds. Moreover, a severe limitation in common design practice is the lack of formal specifications. Requirements are written in languages that are not suitable for mathematical analysis and verification. Assessing system correctness is then left for simulation and, later in the design process, prototyping. Thus, the traditional heuristic design process based on informal requirements and designers' experience can lead to implementations that are inefficient and sometimes do not even satisfy the requirements, yielding long redesign cycles, cost overruns and unacceptable delays.

The cost of being late to market or of product malfunctioning is staggering as witnessed by the recent recalls and delivery delays that system industries had to bear. Toyota's infamous recall of approximately 9 million vehicles due to the sticky accelerator problem[1] and Boeing's 787 delay bringing an approximate toll of \$3.3 billion[2] are examples of devastating effects that design problems may cause. If this is the present situation, the problem of designing planetary-scale CPSs appears insurmountable unless bold steps are taken to bridge the gap between *system science* and *system engineering*.

Several languages and tools have been proposed over the years to overcome the limitations above and enable model-based development of CPSs. However, an all-encompassing framework for CPS design that interconnects different tools, possibly operating on different system representations, is still missing [3]. By reflecting on the history of achievements of electronic design automation in taming the design complexity of VLSI systems, we advocate that CPS design automation efforts are doomed to be impractical and poorly scalable, unless they are framed in *structured design methodologies* and in a formalization of the design process in a *hierarchical* and *compositional* way. Hierarchy has been instrumental to scalable VLSI design, where boosts in productivity have always been associated with a rise in the level of abstraction of design capture, from transistor to register transfer level (RTL), to systems-on-chip. On the other hand, designers typically assemble large and complex systems from smaller and simpler components, such as predesigned intellectual property (IP) blocks. Therefore, compositional approaches offer a "natural" perspective that should inform the whole design process, starting from its earlier stages.

In this paper, we present a path towards an integrated framework for CPS design; the pillars for the framework are a methodology that relies on the *platform-based design* paradigm (PBD) [4] and the algebra of *contracts* to formalize the design process and enable the realization of systems in a hierarchical and compositional manner. Contracts are mathematical abstractions, explicitly defining the assumptions of each component on its environment and the guarantees of the component under these assumptions. The design is carried out as a sequence of refinement steps from a high-level specification to an implementation built out of a library of components at the lower level. The high-level specification is first formalized in terms of contracts to enable requirement validation and early detection of inconsistencies. Then, at each step, contracts are refined by combining synthesis, optimization, and simulation-based design space exploration methods.

We review formalisms and tools that can be used to specify, analyze, or synthesize the design at different levels of abstractions, from the level of discrete systems to the one of hybrid systems. For each formalism, we highlight how the contract operators can be computed and expose the main research challenges for their implementation. We conclude by illustrating our approach on the design of embedded controllers for aircraft electric power distribution systems.

II. PLATFORM-BASED DESIGN WITH CONTRACTS FOR CYBER-PHYSICAL SYSTEMS

We consider in this paper a particular case of CPS that incorporates most, if not all, of the features of general CPSs to help explain the methodology: a *control system*, composed of a physical *plant*, including sensors and actuators, and an embedded *controller*. The controller runs a *control algorithm* to restrict the behaviors of the plant so that all the remaining (closed-loop) behaviors satisfy a set of *system requirements*. Specifically, we consider *reactive controllers*, i.e., controllers that maintain an ongoing interaction with their *environment* by appropriately reacting to it. Our goal is to design the *system architecture*, i.e., the interconnection among system components, and the control algorithm to satisfy the set of high-level requirements.

As shown in Fig. 4(a), the design methodology consists of two main steps, namely, system architecture design and control design. The *system architecture design* step instantiates system components and interconnections among them to generate an optimal architecture while guaranteeing the desired performance, safety, and reliability. Typically, this design step includes the definition of both the embedded system and the plant architectures. The *embedded system architecture* consists of software, hardware, and communication components, while the *plant architecture* depends on the physical system under control, and may consist of mechanical, electrical, hydraulic, or thermal components. Sensors and actuators reside at the boundary between the embedded system and the plant [5]. Given an architecture, the *control design* step

[1]See, e.g., http://www.autorecalls.us.
[2]See, e.g., http://en.wikipedia.org/wiki/Boeing_787.

includes the exploration of the control algorithm and its deployment on the embedded platform.

The above two steps are however connected. The correctness of the controller needs to be enforced in conjunction with the assumptions on the plant. Similarly, performance and reliability of an architecture should be assessed for the plant in closed loop with the controller.

At the highest level of abstraction, the starting point is a set of requirements, predominantly written in text-based languages that are not suitable for mathematical analysis and verification. The result is a model of both the architecture and the control algorithms to be further refined in subsequent design stages. We place this process in the form of Platform-Based Design, and we use contracts extensively to verify the design and to build refinements that are correct by construction.

A. Platform-Based Design

In PBD, at each step, top–down refinements of high-level specifications are mapped onto bottom–up abstractions and characterizations of potential implementations. Each abstraction layer is defined by a design *platform*, which is the set of all architectures that can be built out of a *library* (collection) of *components* according to *composition rules*. In the *bottom–up phase* of each design step, we build and model the component library (including both plant and controller). In the *top–down phase*, we formalize the high-level system requirements and we perform an optimization (refinement) phase called *mapping*, where the requirements are mapped onto the available implementation library components and their composition.

Mapping is cast as an optimization problem, where a set of performance metrics and quality factors are optimized over a space constrained by both system requirements and component feasibility constraints. Mapping is the mechanism that allows to move from a level of abstraction to a lower one using the available components within the library. Note that when some constraint cannot be satisfied using the available library components or the mapping result is not satisfactory for the designer, additional elements can be designed and inserted into the library. For example, when implementing an algorithm with code running on a processor, we are assigning the functionality of the algorithm to a processor and the code is the result of mapping the "equations" describing the algorithm onto the instruction set of the processor. If the processor is too slow, then real-time constraints may be violated. In this case, a new processor has to be found or designed that executes the code fast enough to satisfy the real-time constraint. In the mapping phase, we consider different *viewpoints* (aspects, concerns) of the system (e.g., functional, reliability, safety, timing) and of the components.

If the design process is carried out as a sequence of refinement steps from the most abstract representation of the design platform (top-level requirements) to its most concrete representation (physical implementation), providing guarantees on the correctness of each step becomes essential. Specifically, we seek mechanisms to formally prove that: 1) a set of requirements is *consistent*, i.e., there exists an implementation satisfying all of them; 2) an aggregation of components is *compatible*, i.e., there exists an environment in which they can correctly operate; 3) an aggregation of components *refines* a specification, i.e., it implements the specification and is able to operate in any environment admitted by it. Moreover, whenever possible, we require the above proofs to be performed *automatically* and *efficiently*, to tackle the complexity of today's CPSs. Therefore, to formalize the above design concepts and enable the realization of system architectures and control algorithms in a hierarchical and compositional manner that satisfies the constraints and optimizes the cost function(s), we resort to *contracts*.

B. Contracts: An Overview

The notion of contracts originates in the context of compositional assume-guarantee reasoning [6], which has been used for a long time, mostly for software verification. In a contract framework, design and verification complexity are reduced by decomposing system-level tasks into more manageable subproblems at the component level, under a set of assumptions. System properties can then be inferred or proved based on component properties. Rigorous contract theories have been developed over the years, including assume-guarantee (A/G) contracts [7] and interface theories [8]. However, their concrete adoption in CPS design is still in its infancy, a major challenge being the absence of a comprehensive modeling formalism for CPSs, due to their complexity and heterogeneity [9], [10].

In this paper, we adopt the *assume-guarantee (A/G) contract* framework, as introduced by Benveniste *et al.* [7], [10], to reason about requirements and their refinement during the design process. Because of the explicit distinction between component and environment, A/G contracts are deemed as a rigorous yet intuitive framework, which directly conforms to the thought process of a designer, aiming to guarantee certain performance figures for the design under specific assumptions on its environment. An integration language incorporating A/G contracts to formalize system requirements and enable the generation of simulation monitors has been proposed within the META research program [11] with the aim to compress the product development and deployment timeline of defense systems. Furthermore, over the last few years, many publications have demonstrated the application of A/G contracts in different domains, such as automotive [12], [13], analog integrated systems [5], and, more recently, synthesis and verification of control algorithms for CPSs [14]–[17].

Since A/G contracts are centered around *behaviors*, they are expressive and versatile enough to encompass all kinds of models encountered in system design, from hardware and software models to representations of physical phenomena [10], [18]. The particular structure of the behaviors is defined by specific instances of the contract model. This will only affect the way operators in the contract algebra are implemented since the basic definitions will not vary.

In the sequel, before describing the steps of our methodology, we detail the notions of components and contracts.

C. Components and Contracts

Since PBD is based on the composition of components while refining the design, we start our analysis with a formal representation of a component and associate to it a set of properties that the component satisfies expressed with contracts. The contracts will be used to verify the correctness of the composition and of the refinements.

A *component* M can be seen as an abstraction representing an element of a design, characterized by the following *attributes*:

- A set of input U, output Y, and internal X *variables* (including state variables); a set of configuration *parameters* K, and a set of input, output, and bidirectional *ports* Λ. Components can be connected together by sharing certain ports under constraints on the values of certain variables. In what follows, to simplify, we use the same term variables to denote both component variables and ports.
- A set of *behaviors*, which can be implicitly represented by a dynamic *behavioral model* $\mathcal{F}(u, y, x, \kappa) = 0$, uniquely determining the values of the output $(y \in \mathcal{Y})$ and internal $(x \in \mathcal{X})$ variables given the values of the input variables $(u \in \mathcal{U})$ and configuration parameters $(\kappa \in \mathcal{K})$. Components can respond to every possible sequence of input variables, i.e., they are receptive to their input variables. Behaviors are generic and could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model. In the following, we also use $[\![M]\!]$ to denote the set of behaviors of a component.
- A set of *nonfunctional models*, i.e., maps that allow computing nonfunctional attributes of a component, corresponding to particular valuations of its input variables and configuration parameters. Examples of nonfunctional maps include the *performance model* $\mathcal{P}(.) = 0$, computing a set of performance figures (e.g., bandwidth, latency) by solving a behavioral model, or the *reliability model* $\mathcal{R}(.) = 0$, providing the failure probability of a component.

Components can be hierarchically organized to represent the system at different levels of abstraction. A system can then be assembled by *parallel composition* and interconnection of components at level l, and represented as a new component at level $l+1$. We denote the composition of two components M_1 and M_2, when it is defined, as $M_1 \times M_2$. Then, the behaviors of the composition can be described, in general, as the intersection of the behaviors of its components, i.e., $[\![M_1 \times M_2]\!] = [\![M_1]\!] \cap [\![M_2]\!]$. At each level of abstraction, components are also capable of exposing multiple, complementary *viewpoints*, associated with different design concerns (e.g., safety, performance, reliability) and with models that can be expressed via different formalisms, and analyzed by different tools. Finally, a component M may be associated with a contract, offering a *specification* for it, as further detailed below.

To simplify, in the sequel, we always refer to components with a fixed configuration, i.e., components in which the configuration parameters K are fixed. Then, a *contract* $\mathcal{C}$ for a component M is a triple (V, A, G), where $V = U \cup Y \cup X$ is the set of component variables, and A and G are assertions, each representing a set of behaviors over V [7]. A represents the *assumptions* that M makes on its environment, and G represents the *guarantees* provided by M under the environment assumptions.

A component M satisfies a contract $\mathcal{C}$ whenever M and $\mathcal{C}$ are defined over the same set of variables, and all the behaviors of M satisfy the guarantees of $\mathcal{C}$ in the context of the assumptions, i.e., when $[\![M]\!] \cap A \subseteq G$. We denote this *satisfaction* relation by writing $M \models \mathcal{C}$, and we say that M is an *implementation* of $\mathcal{C}$. However, a component E can also be associated to a contract $\mathcal{C}$ as an *environment*. We say that E is a legal environment of $\mathcal{C}$, and write $E \models_E \mathcal{C}$, whenever E and $\mathcal{C}$ have the same variables and $[\![E]\!] \subseteq A$.

As an example, we consider the amplifier component *Amp* represented in Fig. 1(a), whose amplification gain is two. To specify its operation, we can then formulate a simple (stateless) contract as follows:

$$\mathcal{C}_{\text{amp}} = \big(\{u, y\}, \{(u, y) \in \mathbb{R}^2 |\ |u| \le 1\}, \\ \{(u, y) \in \mathbb{R}^2 |\ y = 2u\}\big)$$

where we use $|u|$ to denote the absolute value of u, and constraints (predicates) on the real variables u and y to characterize the sets of assumptions and guarantees of $\mathcal{C}_{\text{amp}}$. For brevity's sake, when the domain of all the component variables is known, we can also represent assumptions and guarantees directly in terms of predicates, e.g., $\mathcal{C}_{\text{amp}} = (\{u, y\}, |u| \le 1, y = 2u)$, where we implicitly assume that an assumption predicate ϕ_A and a guarantee predicate ϕ_G are both interpreted over the whole set of contract variables. Moreover, A and G will be, respectively, the set of all the behaviors satisfying ϕ_A and ϕ_G.

The component *Amp* duplicates the value of any real number u in the interval $[-1, 1]$, provided as an input. Because the behavior of *Amp* is only determined for a specific input range, there is potentially an infinite number of implementations for $\mathcal{C}_{\text{amp}}$. In particular, a component M_{amp}, defined over the same set of variables $\{u, y\}$, and enforcing $y = 2u$ for all $u \in \mathbb{R}$, is certainly an implementation for $\mathcal{C}_{\text{amp}}$, i.e., $M_{\text{amp}} \models \mathcal{C}_{\text{amp}}$. In fact, its set of behaviors

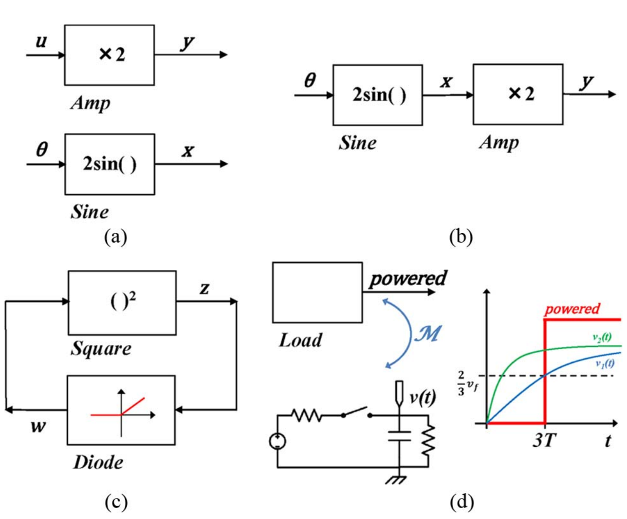


Fig. 1. ***Pictorial representation of the components and interconnections used to illustrate some of the contract operations and relations: (a) parallel composition, (b) serial composition, (c) feedback composition, (d) heterogeneous refinement.***

$[\![M_{\text{amp}}]\!] = \{(u,y) \in \mathbb{R}^2 | y = 2u\}$ coincides with the guarantees of $\mathcal{C}_{\text{amp}}$, and therefore $[\![M_{\text{amp}}]\!] \cap A_{\text{amp}} \subseteq G_{\text{amp}}$ trivially holds. On the other hand, let M'_{amp} be an amplifier with saturation, defined over the same set of variables $\{u,y\}$ and characterized by the following behavioral model:

$$M'_{\text{amp}} : \begin{cases} y = 2u & \forall u \in \mathbb{R} : -1 \le u \le 1 \\ y = -2 & \forall u \in \mathbb{R} : u < -1 \\ y = 2 & \forall u \in \mathbb{R} : u > 1. \end{cases} \tag{1}$$

M'_{amp} blocks its output to a constant value when the magnitude of its input exceeds one. However, in the context of the assumptions A_{amp}, it satisfies G_{amp}; therefore, by definition of contract satisfaction, M'_{amp} is also an implementation of $\mathcal{C}_{\text{amp}}$.

Any component satisfying the assumptions of $\mathcal{C}_{\text{amp}}$ is a legal environment for it; specifically, a component E_{amp} defined over $\{u,y\}$ and providing as an output $u = 0$ for all $y \in \mathbb{R}$ is legal, i.e., $E_{\text{amp}} \models_E \mathcal{C}_{\text{amp}}$. Moreover, given a legal environment E_{amp}, the composition $E_{\text{amp}} \times M_{\text{amp}}$, for all implementations M_{amp}, generates a closed system.

Two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical variables, identical assumptions, and such that $G' \cup \overline{A} = G \cup \overline{A}$, where $\overline{A}$ is the complement of A, possess identical sets of environments and implementations. Such two contracts are then *equivalent*. In particular, any contract $\mathcal{C}$ is equivalent to a contract in *saturated form* $\mathcal{C}'$, obtained by taking $G' = G \cup \overline{A}$. For instance, the contract $\mathcal{C}'_{\text{amp}} = (\{u,y\}, |u| \le 1, |u| \le 1 \rightarrow y = 2u)$ is equivalent to $\mathcal{C}_{\text{amp}}$ since it has the same sets of environments and implementations. However, differently than $\mathcal{C}_{\text{amp}}$, $\mathcal{C}'_{\text{amp}}$ is in saturated form; its set of guarantees $G'_{\text{amp}} = (\{u,y\}, |u| > 1 \vee y = 2u)$ is maximal and coincides with the union of the behaviors of all its implementations. In what follows, we assume that all contracts are in saturated form.

1) *Composition:* Contracts associated to different components can be combined according to different rules. Similar to parallel composition of components, *parallel composition* ($\otimes$) of contracts can be used to construct composite contracts out of simpler ones. Let $\mathcal{C}_1 = (V, A_1, G_1)$ and $\mathcal{C}_2 = (V, A_2, G_2)$ be contracts (in saturated form) over the same set of variables V. The composite contract $\mathcal{C}_1 \otimes \mathcal{C}_2$ is defined as the triple (V, A, G) where

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \tag{2}$$
$$G = G_1 \cap G_2. \tag{3}$$

The composite contract must satisfy the guarantees of both, which explains the operation of intersection in (3) [10]. Intuitively, the assumptions of the composite contract should also be the conjunction of the assumptions of each contract since the environment should satisfy all the assumptions. However, in general, part of the assumptions A_1 will be already satisfied by composing $\mathcal{C}_1$ with $\mathcal{C}_2$, acting as a partial environment for $\mathcal{C}_1$. Therefore, G_2 can contribute to relaxing the assumptions A_1, and vice versa.

As an example, let us consider a simple producer–consumer system, where the producer M_1 is interconnected in series with the consumer M_2, sharing the variable $y \in \mathbb{R}$. Let $\mathcal{C}_1 = (\{y\}, \mathtt{T}, y > 0)$ and $\mathcal{C}_2 = (\{y\}, y > 0, \mathtt{T})$ be the two contracts specifying the behaviors of M_1 and M_2, respectively, both in saturated form. In this example, both assumptions and guarantees are expressed as predicates on y, and $\mathtt{T}$ is the Boolean value True. M_1 guarantees that y is a positive number, which coincides with the assumption made by M_2 on its environment. Then, by applying (2) and (3), we obtain $G = (y > 0)$ and $A = (y > 0) \vee (y \le 0) = \mathtt{T}$, denoting that the composite system is able to operate in any environment, which is intuitive, since the assumptions of M_2 on its environment are relaxed by the guarantees of M_1.

Specifically, when computing (2), we are interested in the maximum set of behaviors A such that $A \cap G_2 \subseteq A_1$ and $A \cap G_1 \subseteq A_2$, where "maximum" refers to the order of sets by inclusion [10]. This is equivalent to finding

$$\begin{aligned} A &= \max\{A' | A' \subseteq A_1 \cup \overline{G_2}, A' \subseteq A_2 \cup \overline{G_1}\} \\ &= (A_1 \cup \overline{G_2}) \cap (A_2 \cup \overline{G_1}) \\ &= (A_1 \cap A_2) \cup (A_1 \cap \overline{G_1}) \cup (A_2 \cap \overline{G_2}) \cup (\overline{G_1} \cap \overline{G_2}) \\ &= (A_1 \cap A_2) \cup \overline{G_1} \cup \overline{G_2} \end{aligned} \tag{4}$$

which reduces to (2). The last equality in (4) stems from the fact that $G = G \cup \overline{A}$ holds for a contract $\mathcal{C} = (V, A, G)$ in saturated form. Contract composition preserves saturated form, that is, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are in saturated form, then so is $\mathcal{C}_1 \otimes \mathcal{C}_2$. Moreover, $\otimes$ is associative and commutative, and generalizes to an arbitrary number of contracts. We therefore can write $\mathcal{C}_1 \otimes \mathcal{C}_2 \otimes \cdots \otimes \mathcal{C}_n$.

For composition to be defined, contracts need to be over the same set of variables V. If this is not the case, then, before composing the contracts, we must first extend their behaviors to a common set of variables using an inverse projection type of transformation, which we call *alphabet equalization*. Formally, let $\mathcal{C} = (V, A, G)$ be a contract and let $V' \supseteq V$ be the set of variables on which we want to extend $\mathcal{C}$. The *extension* of $\mathcal{C}$ on V' is the new contract $\mathcal{C}' = (V', A', G')$ where A' and G' are sets of behaviors over V', defined by inverse projection of A and G, respectively. In the sequel, we freely compose contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ over arbitrary sets of variables V_1, V_2, by implicitly first taking their extensions to $V = V_1 \cup V_2$.

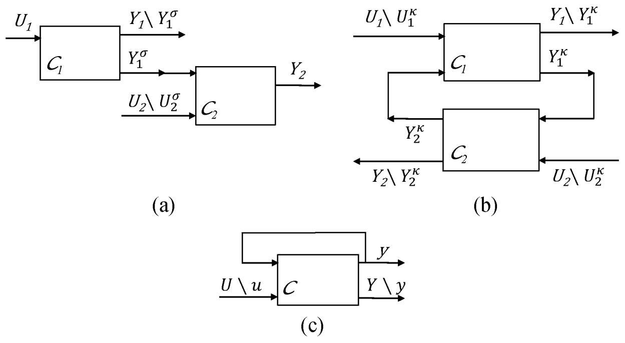
As an example, consider the component *Sine* shown in Fig. 1(a), which receives as input an angle θ and produces an output proportional to the sine of θ. We would like to characterize the contract $\mathcal{C}_{\text{sin}} \otimes \mathcal{C}'_{\text{amp}}$, specifying the parallel composition of *Amp* and *Sine*, where $\mathcal{C}_{\text{sin}} = (\{\theta, x\}, \mathtt{T}, x = 2\sin\theta)$. Moreover, we assume that the components interact by sharing their input variables, which we capture by renaming u as θ. Then, to combine correctly the assumptions and guarantees according to (2) and (3), we first need to extend them to the variable set $\{\theta, x, y\}$, thus obtaining

$$\begin{aligned}\mathcal{C}''_{\text{amp}} &= (\{\theta, x, y\}, |\theta| \le 1, (|\theta| \le 1) \rightarrow (y = 2\theta))\\ \mathcal{C}'_{\text{sin}} &= (\{\theta, x, y\}, \mathtt{T}, x = 2\sin\theta).\end{aligned}$$

Finally, we can compute the assumptions and guarantees of the composite contract as follows:

$$\begin{aligned}G_{\otimes} &:= (x = 2\sin\theta) \wedge ((y = 2\theta) \vee (|\theta| > 1))\\ A_{\otimes} &:= (|\theta| \le 1) \vee (x \ne 2\sin\theta) \vee ((y \ne 2\theta) \wedge (|\theta| \le 1))\\ &= (|\theta| \le 1) \vee (x \ne 2\sin\theta).\end{aligned}$$

As informally introduced by the producer-consumer example above, both *serial* and *feedback compositions* of contracts can be defined using the notion of parallel composition. Feedback composition in the context of contracts has also been investigated in a seminal paper by Benvenuti *et al.* [19]. Let $\mathcal{C}_1 = (V_1, A_1, G_1)$ and $\mathcal{C}_2 = (V_2, A_2, G_2)$ be two contracts, in which the variable sets $V_1 = U_1 \cup Y_1$ and $V_2 = U_2 \cup Y_2$ are, respectively, partitioned into finite sets of input (U_1, U_2) and output variables (Y_1, Y_2).[3] Moreover, we assume that all sets U_1, Y_1, U_2, Y_2 are pairwise disjoint. Then, a *serial interconnection structure* σ, defined as a subset of pairs of $Y_1 \times U_2$, i.e., $\sigma \subseteq Y_1 \times U_2$, generates a renaming on $\mathcal{C}_2$ where, for each pair $(y, u) \in \sigma$, u is renamed as y. Let $Y_1^\sigma = \{y | \exists u : (y, u) \in \sigma\}$ and $U_2^\sigma = \{u | \exists y : (y, u) \in \sigma\}$. As represented in Fig. 2(a), we can then define a *renaming* operator on $\mathcal{C}_2$, $ren_\sigma(\mathcal{C}_2)$, which returns a



U_1
$Y_1 \setminus Y_1^\sigma$
Y_1^σ
$\mathcal{C}_1$
Y_2
$U_2 \setminus U_2^\sigma$
$\mathcal{C}_2$
(a)
$U_1 \setminus U_1^\kappa$
$Y_1 \setminus Y_1^\kappa$
Y_1^κ
Y_2^κ
$Y_2 \setminus Y_2^\kappa$
$U_2 \setminus U_2^\kappa$
(b)
y
$U \setminus u$
$\mathcal{C}$
$Y \setminus y$
(c)

Fig. 2. ***Pictorial representation of different examples of contract compositions: (a) serial composition, (b) feedback composition of two contracts, (c) feedback composition of one contract.***

new contract $\mathcal{C}_2^\sigma = (U_2 \setminus U_2^\sigma \cup Y_1^\sigma \cup Y_2, A_2^\sigma, G_2^\sigma)$, where A_2^σ and G_2^σ are obtained from A_2 and G_2 after renaming their respective variables according to σ. Finally, we can define the *serial composition* of $\mathcal{C}_1$ and $\mathcal{C}_2$ as $\mathcal{C}_1 \overset{\sigma}{\rightsquigarrow} \mathcal{C}_2 := \mathcal{C}_1 \otimes \mathcal{C}_2^\sigma$. For example, we compute the cascade composition of $\mathcal{C}_{\text{sin}} \overset{\sigma}{\rightsquigarrow} \mathcal{C}_{\text{amp}}$ for $\sigma = \{(x, u)\}$, as shown in Fig. 1(b). After renaming and alphabet equalization, by using (2) and (3), we obtain

$$\begin{aligned}G_\sigma &:= (x = 2\sin\theta) \wedge ((y = 2x) \vee (|x| > 1))\\ A_\sigma &:= (|x| \le 1) \vee (x \ne 2\sin\theta)\end{aligned}$$

where both predicates are now to be interpreted on $\{\theta, x, y\}$.

Similarly, a *feedback interconnection structure* κ can be defined as a subset of pairs $\kappa \subseteq (Y_1 \times U_2) \cup (Y_2 \times U_1)$, thus generating a renaming on both $\mathcal{C}_1$ and $\mathcal{C}_2$ where, for each pair $(y, u) \in \kappa$, u is renamed as y. Let $Y_1^\kappa = \{y \in Y_1 | \exists u \in U_2 : (y, u) \in \kappa\}$, $U_2^\kappa = \{u \in U_2 | \exists y \in Y_1 : (y, u) \in \kappa\}$, $Y_2^\kappa = \{y \in Y_2 | \exists u \in U_1 : (y, u) \in \kappa\}$, and $U_1^\kappa = \{u \in U_1 | \exists y \in Y_2 : (y, u) \in \kappa\}$. We can then define a *renaming* operator ren_κ on $\mathcal{C}_1$ and $\mathcal{C}_2$, which returns the new contracts $\mathcal{C}_1^\kappa = (U_1 \setminus U_1^\kappa \cup Y_2^\kappa \cup Y_1, A_1^\kappa, G_1^\kappa)$, and $\mathcal{C}_2^\kappa = (U_2 \setminus U_2^\kappa \cup Y_1^\kappa \cup Y_2, A_2^\kappa, G_2^\kappa)$, as represented in Fig. 2(b). Finally, we can define the *feedback composition* of $\mathcal{C}_1$ and $\mathcal{C}_2$ as $\mathcal{C}_1 \circ_\kappa \mathcal{C}_2 := \mathcal{C}_1^\kappa \otimes \mathcal{C}_2^\kappa$.

A special case of feedback interconnection occurs when a set of outputs of a contract is directly connected to a set of its inputs, as represented in Fig. 2(c). For instance, given a contract $\mathcal{C} = (V, A, G)$, in which $V = U \cup Y$, with U and Y finite sets of input and output variables, and $U \cap Y = \emptyset$, let $\kappa = (y, u) \in Y \times U$ be a *feedback interconnection* on $\mathcal{C}$, connecting an output of $\mathcal{C}$ to one of its inputs, and let

[3]To simplify, we do not explicitly mention the internal variables X of a contract in our discussion of serial and feedback compositions since we can assume, without loss of generality, that X is a subset of the output variables Y.

$\mathcal{C}_{id,\kappa} = (\{y,u\}, \mathtt{T}, y = u)$ a contract that guarantees that the variables supposed to be connected in κ are set to be equal. To simplify, we express the guarantees of $\mathcal{C}_{id,\kappa}$ by using a stateless constraint over its variable set. However, stateful extensions, including temporal constructs, are straightforward. We can then reduce the feedback connection on a contract to the general case of feedback composition defined above by redefining the new contract generated by κ as $\kappa(\mathcal{C}) := \mathcal{C} \circ_\kappa \mathcal{C}_{id,\kappa} = \mathcal{C}^\kappa$. It is then possible to extend the notions of compatibility and consistency of contracts to their serial and feedback compositions as further discussed below.

2) Compatibility and Consistency: $\mathcal{C}$ is *compatible* if there exists a legal environment E for it, i.e., if and only if $A \neq \emptyset$. The intent is that a component satisfying contract $\mathcal{C}$ can only be used in the context of a compatible environment, to be assured that its behaviors conform with the ones specified by the contract. Similarly, a contract is *consistent* when the set of implementations satisfying it is not empty, i.e., it is feasible to develop implementations for it. For contracts in saturated form, this amounts to verify that $G \neq \emptyset$.

When there is a clear distinction between input (*uncontrolled*) and output (*controlled*) variables, different notions of contract compatibility and consistency can be defined [7], [9], [18]. Let $U \subseteq V$ and $Y \subseteq V$ be, respectively, the subset of input and output variables of $\mathcal{C}$, with $U \cap Y = \emptyset$. Then, $\mathcal{C}$ is compatible if and only if A is Y*-receptive*, i.e., if and only if for all behaviors ρ' restricted to variables in Y, there exists a behavior $\rho \in A$, such that ρ' and ρ coincide over Y. Intuitively, an environment has no control on the variables set by an implementation, and therefore A accepts any history offered to the subset Y of its variables. Similarly, $\mathcal{C}$ is consistent if and only if G is X*-receptive*.

Based on these definitions, $\mathcal{C}_{\mathrm{amp}}$ and $\mathcal{C}_{\sin}$ are both compatible and consistent, while a contract $\mathcal{C}_{\mathrm{amp1}} = (\{u,y\}, |u| \leq 2, |u| \leq 1 \rightarrow (y = 2u) \wedge (y > 3))$ is compatible but inconsistent. In fact, compatibility checking amounts to asking whether $\forall y : |u| \leq 2$ is satisfiable, which is true. On the other hand, consistency checking produces $\forall u : |u| \leq 1 \rightarrow (y = 2u) \wedge (y > 3) = \mathtt{F}$ ($\mathtt{F}$ being the Boolean value False), since it is impossible to satisfy the guarantees of $\mathcal{C}_{\mathrm{amp1}}$ for any u in the interval $[-1, 1]$.[4]

The definitions above can be lifted to pairs of contracts, so that $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible (consistent) if and only if $\mathcal{C}_1 \otimes \mathcal{C}_2$ is compatible (consistent). As an example, we show how to derive compatibility and consistency conditions for the cascade of contracts in Fig. 2(a). To be concrete, we assume that the assumptions and guarantees of $\mathcal{C}_1$ and $\mathcal{C}_2$ are represented in terms of predicates or logic formulas on their variables, i.e., $\mathcal{C}_1 = (U_1 \cup Y_1, \phi_{A1}, \phi_{G1})$ and $\mathcal{C}_2 = (U_2 \setminus U_2^\sigma \cup Y_1^\sigma \cup Y_2, \phi_{A2}, \phi_{G2})$. Moreover, since $\mathcal{C}_1$ and $\mathcal{C}_2$ are in saturated form, both $\phi_{A1} \vee \phi_{G1} = \mathtt{T}$ and $\phi_{A2} \vee \phi_{G2} = \mathtt{T}$ must hold. We can then compute assumptions and guarantees for the composite contract $\mathcal{C}_\sigma = \mathcal{C}_1 \overset{\sigma}{\rightsquigarrow} \mathcal{C}_2$ by applying (2) and (3) as follows:

$$\phi_{G\sigma} = \phi_{G1} \wedge \phi_{G2} \tag{5}$$
$$\phi_{A\sigma} = (\phi_{A1} \wedge \phi_{A2}) \vee \neg\phi_{G1} \vee \neg\phi_{G2} \tag{6}$$

where $\phi_{G\sigma}$ and $\phi_{A\sigma}$ must be interpreted as predicates or formulas over the entire set of variables $U_1 \cup Y_1 \cup U_2 \setminus U_2^\sigma \cup Y_2$. In a general case, we would already conclude that $\mathcal{C}_1$ and $\mathcal{C}_2$ are compatible if and only if $\phi_{A\sigma}$ is satisfiable. Similarly, $\mathcal{C}_1$ and $\mathcal{C}_2$ are consistent if and only if $\phi_{G\sigma}$ is satisfiable. However, to gather more insight into compatibility checking, we will further discuss a special case, which is still relevant to several application domains.

We assume that each component of a system only controls its output ports, while the inputs are assigned by the external environment. In this scenario, the assumptions of each contract will only involve its input variables, since the outputs will be under the responsibility of the implementations. Specifically, for the system in Fig. 2(a), ϕ_{A1} and ϕ_{A2} will depend, respectively, only on U_1 and $Y_1^\sigma \cup U_2 \setminus U_2^\sigma$, while ϕ_{G1} and ϕ_{G2} will describe relations involving both the input and output variables of each contract. Finally, we assume that the original contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ are themselves compatible and consistent. In fact, if any contract is incompatible, then it cannot be used in any context; if any contract is inconsistent, it translates into a specification that generates *per se* a contradiction and cannot be implemented.

Our objective is to determine the conditions on the input variables $U_1 \cup U_2 \setminus U_2^\sigma$ that make the composite contract $\mathcal{C}_\sigma$ compatible. To do so, since the variables in $Y_1 \cup Y_2$ cannot be controlled by the environment, we derive new assumptions for $\mathcal{C}_\sigma$ as follows, by using universal quantification over the output variables $Y = Y_1 \cup Y_2$ [18]:

$$\begin{aligned}\phi'_{A\sigma} :=& \forall Y : \phi_{A\sigma}\\ &= \neg\exists Y : (\neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2}) \vee (\neg\phi_{A2} \wedge \phi_{G1} \wedge \phi_{G2})\\ &= (\neg\exists Y : \neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2})\\ &\quad \wedge (\neg\exists Y : \neg\phi_{A2} \wedge \phi_{G1} \wedge \phi_{G2})\\ &= (\neg\exists Y : \neg\phi_{A1} \wedge \phi_{G2}) \wedge (\neg\exists Y : \neg\phi_{A2} \wedge \phi_{G1})\\ &= (\phi_{A1} \vee (\forall Y : \neg\phi_{G2})) \wedge (\forall Y : \phi_{G1} \rightarrow \phi_{A2}).\end{aligned} \tag{7}$$

In the derivations above, we use the fact that quantifiers are commutative and associative to lift them to sets of variables so that $\forall Y := \forall y_1 : \forall y_2 : \ldots : \forall y_n$ when $Y = \{y_1, y_2, \ldots, y_n\}$. Moreover, we leverage the fact that $\neg\phi_{A1} \rightarrow \phi_{G1}$ and $\neg\phi_{A2} \rightarrow \phi_{G2}$ must always hold for contracts in saturated form (implying, e.g., $\neg\phi_{A1} \wedge \phi_{G1} \wedge \phi_{G2} = \neg\phi_{A1} \wedge \phi_{G2}$). Furthermore, since the contrapositive $\neg\phi_{G2} \rightarrow \phi_{A2}$ is also true for $\mathcal{C}_2^\sigma$ (implying $\neg\phi_{G2} = \neg\phi_{G2} \wedge \phi_{A2}$), we obtain

$$(\forall Y : \neg\phi_{G2}) = (\forall Y : \neg\phi_{G2} \wedge \phi_{A2}) = \mathtt{F} \tag{8}$$

[4]In several practical situations, we are interested in contracts that are compatible and consistent at the same time, i.e., satisfying $G \cap A \neq \emptyset$, to discard pathological situations of contracts that are compatible but not consistent, or consistent but not compatible.

since for any input set satisfying ϕ_{A2} there always exists a set of outputs Y_2 that satisfies ϕ_{G2}. Given that ϕ_{A1} does not depend on Y, we can then conclude from (7) that

$$\phi'_{A\sigma} = \phi_{A1} \wedge (\forall Y : \phi_{G1} \rightarrow \phi_{A2}) \tag{9}$$

and that $\mathcal{C}_\sigma$ is compatible if and only if $\phi'_{A\sigma}$ is satisfiable. Intuitively, this is equivalent to require that there exists an environment satisfying the assumptions of $\mathcal{C}_1$, and capable of driving the guarantees of $\mathcal{C}_1$ to become a subset of the assumptions of $\mathcal{C}_2^\sigma$ for any possible assignment of the output variables. For example, we apply the result above to the serial composition $\mathcal{C}_{\text{sin}} \overset{\sigma}{\rightsquigarrow} \mathcal{C}_{\text{amp}}$. Both $\mathcal{C}_{\text{sin}}$ and $\mathcal{C}_{\text{amp}}$ are compatible and consistent contracts. Moreover, $\phi_{A,\text{sin}} = \mathtt{T}$. However, for their composition to be compatible, we also need to enforce $(\forall Y : \phi_{G1} \rightarrow \phi_{A2})$

$$\begin{aligned}
&\forall x : \forall y : (x \neq 2\sin\theta) \vee (|x| \leq 1)\\
&\quad = \neg\exists x : (x = 2\sin\theta) \wedge (|x| > 1)\\
&\quad = \neg\exists x : (x = 2\sin\theta) \wedge (2|\sin\theta| > 1)\\
&\quad = \neg((2|\sin\theta| > 1) \wedge (\exists x : x = 2\sin\theta))\\
&\quad = |\sin\theta| \leq \frac{1}{2} \vee \neg\exists x : (x = 2\sin\theta)\\
&\quad = |\sin\theta| \leq \frac{1}{2}\\
&\quad\Longleftrightarrow \bigvee_{k\in\mathbb{Z}} \left(-\frac{\pi}{6} + k\pi \leq \theta \leq \frac{\pi}{6} + k\pi\right).
\end{aligned}$$

In fact, if θ violates this condition, there is no way for $\mathcal{C}_{\text{sin}}$ to provide a legal environment for $\mathcal{C}_{\text{amp}}^\sigma$.

Compatibility and consistency conditions for the feedback composition in Fig. 2(b) can be determined in a similar way. Computation of the composite contract $\mathcal{C}_\kappa = \mathcal{C}_1^\kappa \circ_\kappa \mathcal{C}_2^\kappa$ generates expressions for the assumptions $\phi_{A\kappa}$ and the guarantees $\phi_{G\kappa}$ that are analogous to (5) and (6). However, in the special case of controlled outputs and uncontrolled inputs, we obtain

$$\phi'_{A\kappa} := \forall Y : \phi_{A\kappa} = \forall Y : (\phi_{G2} \rightarrow \phi_{A1}) \wedge (\phi_{G1} \rightarrow \phi_{A2}) \tag{10}$$

stating that $\mathcal{C}_\kappa$ is compatible if and only if there exists an environment such that, for all possible assignments on the output variables, the guarantees of $\mathcal{C}_1^\kappa$ are included into the assumptions of $\mathcal{C}_2^\kappa$ and vice versa. For example, we investigate the feedback composition of a *Square* component, which squares any input value, with a *Diode* component, which propagates its input to the output only if it is larger or equal to zero, as shown in Fig. 1(c). We assume that the components are formally specified by the contracts $\mathcal{C}_{\text{square}} = (\{w, z\}, \mathtt{T}, z = w^2)$ and $\mathcal{C}_{\text{diode}} = (\{w, z\}, z \geq 0, (z < 0) \vee (w = z))$. Then, since

$$\forall w : \forall z : ((z < 0) \vee (w = z) \rightarrow \mathtt{T}) \wedge (z = w^2 \rightarrow z \geq 0) = \mathtt{T}, \tag{11}$$

we conclude that the two contracts are compatible; moreover, the admitted behaviors can be obtained from the joint guarantees $\phi_{G,\text{square}} \wedge \phi_{G,\text{diode}} := (w = z) \wedge (z = 0 \vee z = 1)$.

Finally, for a contract $\kappa(\mathcal{C}) = (U \cup Y \setminus y, \phi_A, \phi_G)$ as in Fig. 2(c), compatibility checking reduces to verify that

$$\phi'_A := \forall Y : \phi_A = \forall Y : \phi_A \vee \neg\phi_G = \forall Y : \phi_G \rightarrow \phi_A \tag{12}$$

is satisfiable, where we use again the fact that $\neg\phi_G \rightarrow \phi_A$ is always true for a contract in saturated form. The results in this subsection offer a generalization of the conditions for compatibility of contracts in cascade and feedback compositions that were previously reported in the literature [19].

3) Refinement: Refinement is a preorder on contracts, which formalizes a notion of substitutability. We say that $\mathcal{C}$ refines $\mathcal{C}'$, written $\mathcal{C} \preceq \mathcal{C}'$ (with $\mathcal{C}$ and $\mathcal{C}'$ both in saturated form), if and only if $A \supseteq A'$ $(\phi_{A'} \rightarrow \phi_A)$ and $G \subseteq G'$ $(\phi_G \rightarrow \phi_{G'})$. Refinement amounts to relaxing assumptions and reinforcing guarantees, therefore strengthening the contract. Clearly, if $M \models \mathcal{C}$ and $\mathcal{C} \preceq \mathcal{C}'$, then $M \models \mathcal{C}'$. On the other hand, if $E \models_E \mathcal{C}'$, then $E \models_E \mathcal{C}$. We can then replace $\mathcal{C}'$ with $\mathcal{C}$.

As an example, let $\mathcal{C}_{\text{range}} = (\{u, y\}, |u| \leq 1/2, |y| \leq 1)$ be a contract specifying the input and output ranges for the component *Amp*; we would like to show that $\mathcal{C}_{\text{amp}} \preceq \mathcal{C}_{\text{range}}$, that is, when operating in the context of the assumptions of $\mathcal{C}_{\text{range}}$, $\mathcal{C}_{\text{amp}}$ produces an output within the range prescribed by the guarantees of $\mathcal{C}_{\text{range}}$. To do this, we apply the definitions above to the saturated versions of the two contracts; then, refinement checking translates into proving the validity of the following two predicates involving, respectively, the assumptions and the guarantees of both contracts:

$$|u| \leq 1/2 \rightarrow |u| \leq 1 \tag{13}$$

$$(y = 2u) \vee (|u| > 1) \rightarrow (|y| \leq 1) \vee (|u| > 1/2). \tag{14}$$

While (13) is trivially true, to show the validity of (14), we recall that the antecedent in (14) is true when either $(|u| > 1)$ or $(y = 2u)$ holds, and prove that in both cases the consequent is also true. In fact, in the former case, we also have that $(|u| > 1/2)$ holds and the implication is true; in the latter case, if $(1/2 < |u| \leq 1)$ is true, then the implication is still trivially true. If instead $(|u| \leq 1/2)$ is true, we can still conclude $|y| = 2|u| \leq 1$, hence (14) is true.

Alphabet equalization is also needed as a preliminary step to define refinement when $\mathcal{C}$ and $\mathcal{C}'$ are defined over a different alphabet. A more general case of refinement occurs when $\mathcal{C}$ and $\mathcal{C}'$ are also expressed by using different formalisms (*heterogeneous refinement*). In this case, before the refinement relation can be defined, we need to map the behaviors expressed by one of the contracts to the domain of the other contract via a transformation $\mathcal{M}$

(e.g., a type of projection or inverse projection), which is generally more involved than alphabet equalization.

For instance, let $\mathcal{C}_{\text{dis}} = (\{\textit{powered}\}, \mathtt{T}, \diamond_{[0,3)}\textit{powered})$ be the contract specifying the dynamics of a load in an electrical system, which is powered at startup. We will provide details about the temporal construct used to express the guarantees in Section III. For the moment, we point out that $\mathcal{C}_{\text{dis}}$ offers a discrete-time discrete-state abstraction of the dynamics, prescribing that, in all contexts, the Boolean variable *powered* must be asserted within three time units. On the other hand, let $\mathcal{C}_{\text{con}} = (\{v\}, \mathtt{T}, v(t) = v_{\text{f}}(1 - \mathrm{e}^{-(t/\tau)}), t \in \mathbb{R}, t \geq 0)$ be the contract describing the voltage level of the electrical load as a continuous function of time t. The load responds as a first-order dynamical system with time constant τ and steady-state voltage v_{f}. Then, we can reason about refinement between $\mathcal{C}_{\text{con}}$ and $\mathcal{C}_{\text{dis}}$ only if we provide a mechanism to map continuous time and voltage levels into discrete ones. In this case, given a time-step T, such a mechanism could be provided by the following transformation $\mathcal{M}$:

$$\mathcal{M} : \begin{cases} \textit{powered} := \left(v \geq \frac{2}{3}v_{\text{f}}\right) \\ k := \left\lfloor \frac{t}{T} \right\rfloor \end{cases} \tag{15}$$

stating that *powered* is asserted if and only if the voltage is greater than or equal to two thirds of the steady-state value, while the discrete time index k is obtained by discretizing t according to the quantization step T. Resting on this mapping, we can then conclude that $\mathcal{C}_{\text{con}} \preceq_{\mathcal{M}} \mathcal{C}_{\text{dis}}$ if and only if $v(3T) > (2/3)v_{\text{f}}$, i.e., if and only if the system time constant satisfies $\tau < 3T/\ln 3$. This condition is illustrated in Fig. 1(d), where $v_2(t)$ (in green) satisfies the constraint on τ and refines the guarantees of $\mathcal{C}_{\text{dis}}$, whereas $v_1(t)$ (in blue) does not since it reaches the desired value $(2/3)v_{\text{f}}$ exactly at time $t = 3T$ $(k = 3)$, while the interval in the guarantees of $\mathcal{C}_{\text{dis}}$ is right-open.

4) Conjunction: To compose multiple requirements on the *same component*, possibly representing different viewpoints that need to be satisfied simultaneously, we can also define the *conjunction* $(\wedge)$ of contracts. Let $\mathcal{C}_1 = (V, A_1, G_1)$ and $\mathcal{C}_2 = (V, A_2, G_2)$ be contracts (in saturated form) over the same set of variables V and on the same component M. We would like to combine $\mathcal{C}_1$ and $\mathcal{C}_2$ into a joint contract $\mathcal{C}_1 \wedge \mathcal{C}_2$ so that, if $M \models \mathcal{C}_1 \wedge \mathcal{C}_2$, then $M \models \mathcal{C}_1$ and $M \models \mathcal{C}_2$. We can compute the conjunction of $\mathcal{C}_1$ and $\mathcal{C}_2$ by taking their greatest lower bound with respect to the refinement relation, i.e.: 1) $\mathcal{C}_1 \wedge \mathcal{C}_2$ is guaranteed to refine both $\mathcal{C}_1$ and $\mathcal{C}_2$; and 2) for any contract $\mathcal{C}'$ such that $\mathcal{C}' \preceq \mathcal{C}_1$ and $\mathcal{C}' \preceq \mathcal{C}_2$, we have $\mathcal{C}' \preceq \mathcal{C}_1 \wedge \mathcal{C}_2$. For contracts in saturated form and on the same alphabet, we have

$$\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \cup A_2, G_1 \cap G_2). \tag{16}$$

As an example, let $\mathcal{C}_{\text{range1}}$ and $\mathcal{C}_{\text{range2}}$ be two contracts restricting the input and output ranges of an *Amp* component, and defined as follows:

$$\begin{aligned} \mathcal{C}_{\text{range1}} &= (\{u, y\}, 0 \leq u \leq 1/2, 0 \leq u \leq 1/2 \rightarrow y \geq u) \\ \mathcal{C}_{\text{range2}} &= (\{u, y\}, 0 \leq u \leq 1, 0 \leq u \leq 1 \rightarrow 0 \leq y \leq 3u). \end{aligned}$$

Then, we can compute the conjunction $\mathcal{C}_{\text{range1}} \wedge \mathcal{C}_{\text{range2}}$ as

$$\begin{aligned} A_\wedge :=& (0 \leq u \leq 1/2) \vee (0 \leq u \leq 1) = 0 \leq u \leq 1 \\ G_\wedge :=& (0 \leq u \leq 1/2 \rightarrow y \geq u) \wedge (0 \leq u \leq 1 \rightarrow 0 \leq y \leq 3u) \\ =& (u \leq y \leq 3u) \vee (u > 1/2 \wedge 0 \leq y \leq 3u) \vee (u < 0) \\ & \vee (u > 1). \end{aligned}$$

Since $\mathcal{C}_{\text{amp}}$ admits a larger set of inputs, the whole interval $[-1, 1]$, and promises $y = 2u$ for $u \in [0, 1]$, it clearly refines the conjunction contract, hence it refines both $\mathcal{C}_{\text{range1}}$ and $\mathcal{C}_{\text{range2}}$. Therefore, any implementation of $\mathcal{C}_{\text{amp}}$, such as M_{amp}, will also implement both $\mathcal{C}_{\text{range1}}$ and $\mathcal{C}_{\text{range2}}$.

Another form for A/G contracts has also been proposed, which supports reasoning about complex component interactions by avoiding using parallel composition of contracts to overcome the problems that certain models have with the effective computation of the operators [20]. Instead, composition is replaced with the concept of *circular reasoning* [21]. When circular reasoning is sound, it is possible to check relations between composite contracts based on their components only, without taking expensive compositions. However, the notions of compatibility and conjunction, as described above, are not addressed in this theory.

5) Horizontal and Vertical Contracts: Traditionally contracts have been used to specify components and aggregation of components at the *same level of abstraction*, as illustrated by the above examples; for this reason, we refer to them as *horizontal contracts*.

We use contracts also to formalize and reason about refinement between two different abstraction levels in the PBD process [5], [10]; for this reason, we refer to this type of contracts as *vertical contracts*. To illustrate this concept, consider the problem of mapping a specification platform of a system at level $l + 1$ into an implementation platform at level l. In general, the specification platform architecture (i.e., interconnection of components) may be defined in an independent way, and may not directly match the implementation platform architecture. Such a different architectural decomposition will also reflect on the contracts associated with the components and their aggregations. For instance, the contract describing the specification platform $\mathcal{C} = \bigwedge_{k \in K}(\bigotimes_{i \in I_k} \mathcal{C}_{ik})$ may be defined as the conjunction of K different viewpoints, each characterized by its own architectural decomposition into I_k contracts. On the other hand, the contract describing the implementation platform $\mathcal{M} = \bigotimes_{j \in J}(\bigwedge_{n \in N_j} \mathcal{M}_{jn})$ may be better represented as a composition of J contracts, each defined out of a conjunction of its different viewpoints. Because there may not be, in general, a direct matching between contracts and viewpoints of $\mathcal{M}$ and $\mathcal{C}$, checking that $\mathcal{M} \preceq \mathcal{C}$ in a compositional way, by reasoning on the elements of $\mathcal{M}$ and $\mathcal{C}$ independently, as discussed in Section II-C3, may not be effective.

However, it is still possible to reason about refinement between $\mathcal{M}$ and $\mathcal{C}$ by resorting to a contract that specifies the composition of a model and its vertical refinement, even though they are not directly connected, by connecting them indirectly through a mapping, e.g., by synchronizing pairs of events, as if co-simulating a model and its refinement. Informally, this kind of composition captures the fact that the actual satisfaction of all the design requirements and viewpoints by a deployment depends on the supporting execution platform, the underlying physical system, and on the way in which system functionalities are mapped to them. Formally, this composition can be modeled using two alternative methods, based on the specific shapes of $\mathcal{C}$ and $\mathcal{M}$:

- The interaction between the specification and the implementation platforms can be modeled using the contract composition $\mathcal{C} \otimes \mathcal{M}$. In this case, assumptions made by the specification platform on the implementation platform get discharged by the guarantees of the implementation platform, and vice versa, as indicated by (2) and (3). Refinement can then be checked by checking that $\mathcal{C} \otimes \mathcal{M}$ is compatible, and that $\mathcal{C} \otimes \mathcal{M} \preceq \mathcal{C}$, which can be performed compositionally, by matching the elements of $\mathcal{C}$ with the ones of $\mathcal{C} \otimes \mathcal{M}$.
- The interaction between the specification and the implementation platforms can also be modeled using the contract conjunction $\mathcal{C} \wedge \mathcal{M}$. In this case, assumptions and guarantees combine as in (16), and $\mathcal{C} \wedge \mathcal{M}$ is assured to refine $\mathcal{C}$ by construction. However, being a conjunction, it can still be a source of inconsistencies. Therefore, to guarantee that the design can be implemented, the consistency of $\mathcal{C} \wedge \mathcal{M}$ must be checked or enforced by the designer.

Composite contracts such as $\mathcal{C} \otimes \mathcal{M}$ and $\mathcal{C} \wedge \mathcal{M}$ are both called *vertical contract* and can be used to formalize mechanisms for mapping a specification over an execution platform, such as the ones adopted in the METROPOLIS [22], METROII [23], and, more recently, the METRONOMY frameworks [24].

We exemplify the use of vertical contracts by referring to the virtual model of a simple system pictured in Fig. 3. The specification platform architecture, at the top of the figure, consists of two interconnected components. At startup, the *Controller* interacts with an external subsystem through its *in* and *out* ports to perform some high-priority task. Then, it switches on a safety-critical electric power system *EPS*, by asserting its output *on*, and makes sure that the system is actually powered, i.e., the signal *powered* is asserted, by the deadline t_d.

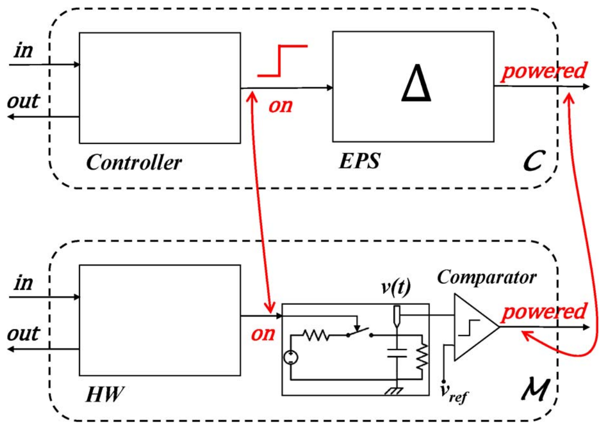


Fig. 3. ***Specification and implementation platform examples used to illustrate vertical contracts.***

At the application level, to conveniently explore different control strategies, the designer abstracts the physical system *EPS* using a simple delay block, which propagates the value of its input *on* to its output *powered* with a delay Δ. We therefore obtain $t_{\mathrm{pow}} - t_{\mathrm{on}} = \Delta$, where t_{pow} and t_{on} are, respectively, the times at which *powered* and *on* are asserted, and Δ is selected to accommodate the delay of the physical platform. Then, the designer implements the required functionality by allocating the *Controller* to its higher priority task, while guaranteeing a worst-case switch-on time $t_{\mathrm{on}}^{\max} = t_{\mathrm{d}} - \Delta$ to meet the deadline on the *powered* signal.

While the functional platform described above is very convenient to explore different control strategies, it is not sufficient to determine the correctness of the final design. In fact, the satisfaction of the timing viewpoints heavily relies on the assumptions on the delay of the physical system, which can only be discharged by the implementation platform. The architecture of the implementation platform is shown at the bottom of Fig. 3. The functionality of the *Controller* is mapped to a hardware execution platform *HW*, while the *EPS* is modeled by a cascade of a first-order filter with time constant τ, represented in the figure as an electrical network, and an ideal *Comparator* block, with reference voltage v_{ref}. If the filter output voltage v is larger than v_{ref}, the *Comparator* asserts its output *powered*. The reference v_{ref} corresponds to 90% of the final value v_{f} reached by v at steady state.

To show that the implementation platform refines the specification platform, hence satisfies the system requirements, we can formalize the interaction between the two levels in terms of the composition $\mathcal{C}^t \otimes \mathcal{M}^t$ between two timing contracts:

- $\mathcal{C}^t = (\{\delta_{\mathrm{on}}, t_{\mathrm{on}}, t_{\mathrm{pow}}\}, \delta_{\mathrm{on}} \leq \Delta,\ (t_{\mathrm{on}} \leq t_{\mathrm{d}} - \Delta) \rightarrow (t_{\mathrm{pow}} \leq t_{\mathrm{d}}))$ specifies the timing behavior of the specification platform, by emphasizing its *vertical assumptions* on the implementation platform. If the implementation platform provides a delay $\delta_{\rm on}$ less than or equal to Δ when *on* is asserted, then the application guarantees to satisfy the requirement on the *powered* signal, if *on* is asserted by at least an interval Δ before the deadline t_d.

- $\mathcal{M}^t = (\{\delta_{\rm on}, t_{\rm on}, t_{\rm pow}\}, \ \mathtt{T}, (\delta_{\rm on} = \tau \ln 10) \wedge (t_{\rm pow} = t_{\rm on} + \delta_{\rm on}))$ exposes the timing behavior of the implementation platform, as derived from the step response of a first-order filter. $\mathcal{M}^t$ states that, whenever *on* is asserted, the delay at which v reaches 90% of its steady-state level, hence $t_{\rm pow}$ is asserted, is $\delta_{\rm on} = \tau \ln 10$.

In this simple example, the assumptions and guarantees of both $\mathcal{C}^t$ and $\mathcal{M}^t$ are assertions over variables denoting the time of occurrence of certain events, or their separation.

Resting on the above contracts, because the assumptions of $\mathcal{C}^t$ trivially imply the ones of $\mathcal{M}^t$, we obtain $\mathcal{C}^t \otimes \mathcal{M}^t \preceq \mathcal{C}^t$. Moreover, for the contracts in this example, it is also possible to show that $\mathcal{M}^t \preceq \mathcal{C}^t \otimes \mathcal{M}^t$ holds. Therefore, checking the correctness of the design finally requires checking that the vertical contract $\mathcal{C}^t \otimes \mathcal{M}^t$ is compatible. In this case, by applying (9) in Section II-C-2, where $\mathcal{M}^t$ and $\mathcal{C}^t$ act, respectively, as $\mathcal{C}_1$ and $\mathcal{C}_2$, we conclude that it is enough to check the satisfiability of $\forall \delta_{\rm on} : \forall t_{\rm pow} : (\delta_{\rm on} = \tau \ln 10) \wedge (t_{\rm pow} = t_{\rm on} + \delta_{\rm on}) \rightarrow (\delta_{\rm on} \leq \Delta)$, which provides

$$\tau \ln 10 \leq \Delta. \tag{17}$$

This inequality can also be used at design time, as a practical guideline to dimension either the specification platform, by increasing its margin Δ, or the implementation platform, by decreasing its time constant τ, to deploy a correct design.

It is also possible to obtain the same result as above, by following an alternative formulation based on contract conjunction. In particular, we suppose that the designer chooses instead to describe the timing behaviors of the system in Fig. 3 using a different contract pair:

- $\tilde{\mathcal{C}}^t = (\{t_{\rm on}, t_{\rm pow}\}, t_{\rm on} \leq (t_d - \Delta), t_{\rm pow} \leq t_d)$, the specification contract, is no longer bound to the implementation platform. It simply states that the requirement on $t_{\rm pow}$ is satisfied if *on* is asserted by at least an interval Δ before the deadline t_d.
- $\tilde{\mathcal{M}}^t = (\{t_{\rm on}, t_{\rm pow}\}, \mathtt{T}, t_{\rm pow} = t_{\rm on} + \tau \ln 10)$, the implementation contract, is also independent of the specification platform (except for being defined on the same variables) and exposes the timing behavior of the *powered* signal. $\tilde{\mathcal{M}}^t$ states that, whenever *on* is asserted, *powered* will be asserted with a delay $\tau \ln 10$, due to the physical system (a first-order filter).

Then, to check the correctness of the refinement, a binding mechanism between the two contracts, each linked to its own platform, can now be provided by the conjunction of $\tilde{\mathcal{M}}^t$ and $\tilde{\mathcal{C}}^t$. $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$ ensures that both contracts are jointly satisfied and refines $\tilde{\mathcal{C}}^t$ by construction. Therefore, all we need to check is that $\tilde{\mathcal{M}}^t$ does not create inconsistencies in $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$, in the sense that $(\forall t_{\rm on} : \exists t_{\rm pow} : G_{\tilde{\mathcal{M}}^t} \cap G_{\tilde{\mathcal{C}}^t})$ is true,[5] where $G_{\tilde{\mathcal{M}}^t}$ and $G_{\tilde{\mathcal{C}}^t}$ are the guarantees of the two contracts in saturated form. In our case

$$\begin{aligned}
&\forall t_{\rm on} : \exists t_{\rm pow} : (t_{\rm pow} = t_{\rm on} + \tau \ln 10) \\
&\qquad\qquad \wedge \left((t_{\rm on} > t_d - \Delta) \vee (t_{\rm pow} \leq t_d)\right) \\
&\quad = \forall t_{\rm on} : (\exists t_{\rm pow} : t_{\rm pow} = t_{\rm on} + \tau \ln 10) \wedge (t_{\rm on} > t_d - \Delta) \\
&\qquad \vee (t_{\rm on} \leq t_d - \tau \ln 10) \\
&\quad = \forall t_{\rm on} : (t_{\rm on} > t_d - \Delta) \vee (t_{\rm on} \leq t_d - \tau \ln 10)
\end{aligned}$$

leads to the condition $\tau \ln 10 \leq \Delta$, which is the result found in (17). Intuitively, this amounts to requiring that, if $t_{\rm on}$ and $t_{\rm pow}$ have to synchronize so that $\tilde{\mathcal{M}}^t$ refines $\tilde{\mathcal{C}}^t$ and the overall system satisfies the timing requirement on $t_{\rm pow}$, then the delay implemented by the physical system in $\tilde{\mathcal{M}}^t$ must be smaller than or equal to the one defined by the application platform in $\tilde{\mathcal{C}}^t$.

The approach illustrated above has been previously adopted in the design of analog and mixed-signal integrated circuits [5], [25] by leveraging effective approximations of implementation constraints to formulate vertical contracts representing different viewpoints (e.g., timing, energy, noise), and then checking their compatibility or consistency during design space exploration. More recently, a similar approach has also been advocated in the context of AUTOSAR [10]. Alternatively, when vertical assumptions and guarantees cannot be effectively expressed by compact models, compatibility and consistency of vertical contracts can be checked by co-simulation of the application and implementation platforms under a mapping mechanisms, such as the one in the METRONOMY framework [24], in which tuples of signals in the two platforms are synchronized. In the context of our example, this technique can be applied by unifying both occurrences and values of the *on* and *powered* signals, as shown in red in Fig. 3, and then checking that the synchronized models satisfy the requirements.

We finally observe that the formal notion of vertical contracts we have presented is general, in that it encompasses other notions of contracts that were previously introduced in a control setting to capture the interactions between the controller and its execution platform [9], [26]. In this scenario, a controller takes as assumptions several aspects that include the timing behavior of the control tasks and of the communication between tasks, e.g., delay, jitter, as well as the accuracy and resolution of the computation (vertical assumptions in $\mathcal{C}$). On the other hand, the

[5]We are actually interested in checking consistency $\forall t_{\rm on} : t_{\rm on} \leq (t_d - \Delta)$, which is the set of legal environments for $\tilde{\mathcal{C}}^t$. In fact, we want to show that, for each $t_{\rm on}$ satisfying the assumptions of the specification contract $\tilde{\mathcal{C}}^t$, there exists an implementable $t_{\rm pow}$, according to the implementation contract $\tilde{\mathcal{M}}^t$, which also satisfies the deadline t_d, as required by $\tilde{\mathcal{C}}^t$. When $t_{\rm on} > (t_d - \Delta)$, $\tilde{\mathcal{C}}^t \wedge \tilde{\mathcal{M}}^t$ is trivially consistent since the guarantees of $\tilde{\mathcal{C}}^t$ are vacuously true.

controller provides guarantees in terms of the amount of requested computation, activation times, and data dependencies (vertical guarantees in $\mathcal{C}$). Vertical contracts can then be effectively used to formalize the agreement between control, software, and hardware engineers, when specifying both system functionality and timing requirements. As a result, several design approaches and guidelines, which have been previously established in the literature in terms of "design contracts" [26], can be derived by formulating vertical contracts for both the software and control layers, and by enforcing their compatibility and consistency as illustrated by the example in Fig. 3.

III. REQUIREMENT FORMALIZATION AND VALIDATION USING CONTRACTS

We use contracts to formalize top-level requirements, allocate them to lower-level components, and analyze them for early validation of design constraints. Requirement analysis can often be challenging because of the lack of familiarity with formal languages among system engineers. Moreover, it is significantly different from traditional formal verification, where a system model is compared against a set of requirements. Since there is not yet a system at this stage, requirements themselves are the only entity under analysis. By formalizing requirements as contracts, it is instead possible to provide effective tests to check for requirement consistency, i.e., whether a set of contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible. Moreover, it is possible to exclude undesired behaviors, e.g., by adding more contracts, by strengthening assumptions, or by considering additional cases for guarantees. Since contracts are abstractions of components, their concrete representations are typically more compact than a fully specified design [15]. The above tests can then be performed more efficiently than traditional verification tasks.

A framework for requirement engineering has been recently developed by leveraging modal interfaces, an automata-based formalism, as the underlying specification theory [10]. However, to retain a correspondence between informal requirements and formal statements, *declarative*, "property-based" approaches using some temporal logic are gaining increasing interest. They contrast *imperative*, "model-based" approaches, which tend to be impractical for high-level requirement validation. In fact, constructing a model to capture all the behaviors allowed by the requirements often entails considering all possible combinations of system variables. Moreover, these models are usually hard to parametrize, and small changes in the requirements become soon hard to map into changes in the corresponding models.

In this paper, we follow an approach based on A/G contracts as introduced in Section II-B, which allows specifying different kinds of requirements using different formalisms, following both the declarative and imperative styles, to reflect the different viewpoints and domains in a heterogeneous system, as well as the different levels of abstraction in the design flow. As shown in Fig. 4(a), to facilitate reasoning at the level of abstraction of requirement engineers, a viable strategy is to drive engineers towards capturing requirements in a structured form, using a set of predefined high-level *primitives*, or patterns, from which formal specifications can be automatically generated. This approach is similar to the one advocated in the STATEMATE verification environment [27], within the European projects SPEEDS and CESAR [13] (linked to automata-based formalisms), or to the higher-level *domain-specific language* (DSL) exemplified in Section VI [14].

From a set of high-level primitives, different kinds of contracts can be generated. When specifying the *system architecture*, steady-state (static) requirements, interconnection rules, and component dimensions can be captured by static contracts, expressed via *arithmetic constraints on Boolean and real variables* to model, respectively, discrete and continuous design choices. Then, compatibility, consistency, and refinement checking translate into checking feasibility of conjunctions or disjunction of constraints, which can be solved via queries to Satisfiability Modulo Theory (SMT) solvers [28], [29] or mathematical optimization software, such as mixed integer-linear, mixed integer-semidefinite-positive, or mixed integer/nonlinear program solvers.

When specifying the *control algorithm*, representing dynamic behaviors becomes the main concern; safety and real-time requirements can then be captured by contracts expressed using *temporal logic* constructs. For instance, linear temporal logic (LTL) [30] can be used to reason about the temporal behaviors of systems characterized by Boolean, discrete-time signals, or sequences of events [discrete event abstraction in Fig. 4(a)]. Signal temporal logic (STL) [31] can deal with dense-time real signals and continuous dynamical models [continuous abstraction in Fig. 4(a)]. Sometimes, discrete and continuous dynamics are so tightly connected that a discrete-event (DE) abstraction would result inaccurate, while a continuous abstraction would turn out to be inefficient, thus calling for a *hybrid system* abstraction, mixing discrete and continuous behaviors, such as Hybrid Linear Temporal Logic with Regular Expressions (HRELTL) [32] and *hybrid automata* [33]. In the sequel, we review the main formalisms for the specification of dynamical systems, and the related tools, which can be used to implement the algebra of contracts and perform requirement analysis within our framework.

A. Temporal Logic

Temporal logic is a symbolism for representing and reasoning about the evolution of a system over time. Starting from the 1980s, it has been successfully applied in formal verification, and a flourishing family of temporal logics has been developed both by academy and industry. Because of its "declarative" flavor, temporal logic seems a

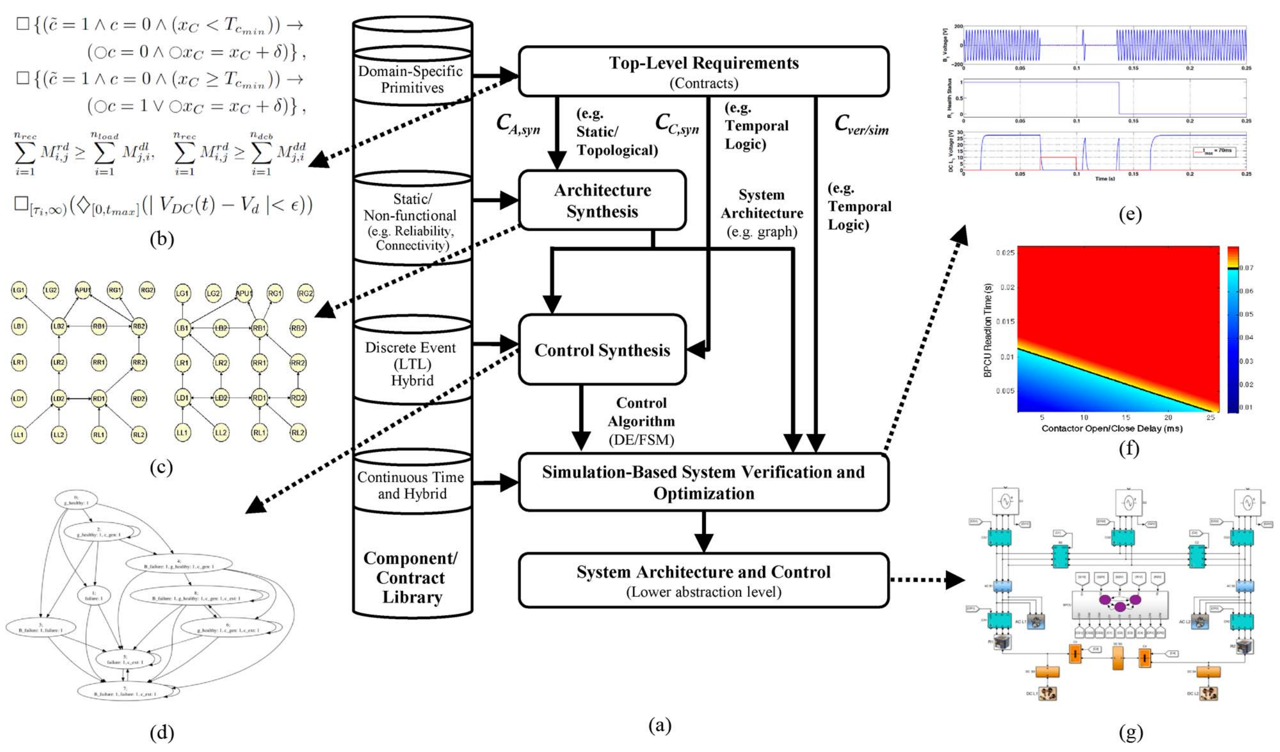

$\Box\{(\tilde{c}=1 \wedge c=0 \wedge (x_C < T_{c_{min}})) \rightarrow (\bigcirc c = 0 \wedge \bigcirc x_C = x_C + \delta)\}$,
$\Box\{(\tilde{c}=1 \wedge c=0 \wedge (x_C \geq T_{c_{min}})) \rightarrow (\bigcirc c = 1 \vee \bigcirc x_C = x_C + \delta)\}$,
$\sum_{i=1}^{n_{rec}} M_{i,j}^{rd} \geq \sum_{i=1}^{n_{load}} M_{j,i}^{dl}$, $\sum_{i=1}^{n_{rec}} M_{i,j}^{rd} \geq \sum_{i=1}^{n_{dcb}} M_{j,i}^{dd}$
$\Box_{[\tau_i,\infty)}(\Diamond_{[0,t_{max}]}(\mid V_{DC}(t) - V_d \mid < \epsilon))$
(b)
(c)
(d)
Domain-Specific Primitives
Static/ Non-functional (e.g. Reliability, Connectivity)
Discrete Event (LTL) Hybrid
Continuous Time and Hybrid
Component/ Contract Library
Top-Level Requirements (Contracts)
$C_{A,syn}$
(e.g. Static/ Topological)
$C_{C,syn}$
(e.g. Temporal Logic)
$C_{ver/sim}$
Architecture Synthesis
System Architecture (e.g. graph)
(e.g. Temporal Logic)
Control Synthesis
Control Algorithm (DE/FSM)
Simulation-Based System Verification and Optimization
System Architecture and Control (Lower abstraction level)
(a)
(e)
BPCU Reaction Time (s)
Contactor Open/Close Delay (ms)
(f)
(g)

Fig. 4. *(a) Structure of the proposed contract-based methodology for CPS design, from top-level requirements to the definition of system architecture and control algorithm. Demonstration of the different design steps on the aircraft electric power system example in Section VI [14]: (b) requirement formalization; (c) plant architecture selection; (d) reactive control synthesis; (e) simulation-based verification; (f) simulation-based design exploration; (g) hybrid power system model in Simulink for further refinement.*

"natural" language to formalize high-level requirements in terms of contracts. Moreover, especially for discrete-time, discrete-state system representations, the wealth of results and tools in temporal logic and *model checking* can provide a substantial technological basis for requirement analysis [6].

Classical *discrete-time temporal logics* like LTL and computation tree logic (CTL) [6], originally developed to state requirements of hardware and software electronic systems, can indeed be effectively used to describe the DE abstraction of CPSs. As an example, in the abstraction offered by LTL, a component can be represented as a set of Boolean variables S_{DE}. Then, the behaviors of a component can be described by the infinite sequences of states of the form $\sigma = s_0 s_1 s_2 \ldots$ satisfying an LTL formula, each state s being a valuation of the Boolean variables in S_{DE}. A sample requirement expressible by LTL is the property "An alert must be eventually resolved," which can be formalized by the formula $\Box\ (alert \rightarrow \Diamond\ sys\_ok)$, where *alert* and *sys_ok* are Boolean component variables. This formula states that every occurrence of the *alert* event (i.e., when *alert* is asserted), as denoted by the *always* ($\Box$) operator, must *eventually* ($\Diamond$) be followed by an occurrence of a *sys_ok* event.

Discrete-time temporal logics, however, lack the expressiveness needed to capture the continuous aspects of the system in a faithful way. To overcome this limitation, temporal logics have been extended in many ways. A first extension, routinely used in the verification of discrete-time hybrid systems, is to replace Boolean variables with first-order atoms, including nonlinear arithmetic constraints on real numbers [34]. In this way, LTL can express properties like "If the temperature reaches 90°, then it must eventually decrease below 60," using formulas of the form $(t \geq 90 \rightarrow \Diamond\ t < 60)$, which constrains any state where the temperature t is greater or equal to 90 to be followed by a state where the temperature is below 60.

A second possibility is to add operators to express timing constraints between discrete events. This leads to the development of *real-time temporal logics* such as Metric Temporal Logic (MTL) [35]. For instance, real-time temporal logics can express properties like "An alert must be resolved in 10 s," by means of the MTL formula $\Box(alert \rightarrow \Diamond_{[0,10]}\ sys\_ok)$, which forces the *sys_ok* event to occur at most 10 time units after the *alert* event.

Real-time temporal logics have been further extended by providing a continuous notion of time and by making them capable of expressing properties of continuous quantities. The most relevant language in this family of *continuous-signal logics* is STL [31], which combines first-order atoms with timing constraints and is able to express properties like "If the temperature reaches 90°, then it must decrease below 60 in at most 10 s." Such a property can be formalized by the formula

$\square(t \geq 90 \rightarrow \diamond_{[0,10]}\, t < 60)$, which constrains any time instant τ_0 where the temperature t is greater or equal to 90 to be followed by a time instant τ_1 where the temperature is below 60 and such that $\tau_1 - \tau_0 \leq 10$.

More recently, some *logics for hybrid-systems* have been introduced, which can express properties of both the discrete and continuous behaviors of a system. Two relevant members of this class are HRELTL [32], which extends the LTL with regular expressions (RE), and Differential Dynamic Logic ($\mathsf{d}\mathcal{L}$) [36], which can specify correctness properties for hybrid systems given operationally as hybrid programs. An example of a hybrid property is "If the temperature reaches 90, then an alert is raised," which can be formalized by the HRELTL formula ($t \geq 90 \rightarrow \bigcirc$ *alert*), where $\bigcirc$ is the "next discrete event" operator. On the other hand, the hybrid property "for the state of a train controller *train*, $z \leq 100$ always holds true when starting in a state where $v^2 \leq 10$ is true,'' can be expressed by the $\mathsf{d}\mathcal{L}$ formula $v^2 \leq 10 \rightarrow [train]z \leq 100$, where z and v are the position and the velocity of the train, respectively.

Temporal Logic and Contracts: Consistent with the representation of component behaviors, both assumptions A and guarantees G of a contract $\mathcal{C}$ can be specified as temporal logic formulas [18]. In this case, a component M satisfies the contract $\mathcal{C}$ if it satisfies the logical implication $A \rightarrow G$, while it is a legal environment for $\mathcal{C}$ if it satisfies the formula A. Contract satisfaction can thus be reduced to two specific instances of model checking [6]. Composition and conjunction of contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ can be represented by appropriate Boolean combination of the formulas A_1, A_2, G_1, and G_2. Other operations on contracts, as defined in Section II-C, can be reduced to special instances of the validity or satisfiability checking problem for temporal logic (or quantified temporal logic [18]), as follows:

- In its simplest formulation, *compatibility and consistency* can be checked by testing whether A or G are *satisfiable*. More complex instances of the problem, which rule out contracts that are "trivially" compatible or consistent, can be solved by *vacuity checking* [37].
- *Refinement* is an instance of *validity checking*: $\mathcal{C}_1 \preceq \mathcal{C}_2$ if and only if $A_1 \rightarrow A_2$ and $G_2 \rightarrow G_1$ are valid formulas (i.e., tautologies for the language).

A solution of the above problems for HRELTL, based on SMT techniques, can be found in [32].

B. Hybrid Automata

Formalisms following an imperative style, such as hybrid automata, can be used to specify functional requirements especially for system portions of limited complexity. For example, describing the intended behavior of a controlled continuous system together with its discrete controller. Then, one can verify the intended behavior versus generic properties such as *safety*, which requires the automata to stay away from a set of "bad" states, as well as

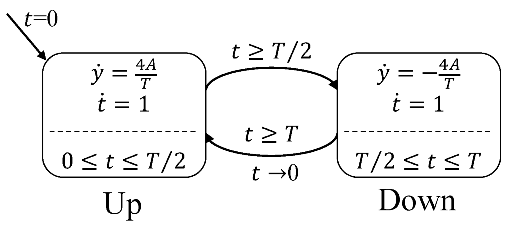


Fig. 5. ***Hybrid automaton specifying a triangle wave generator.***

verify whether an implementation is a refinement of the hybrid automaton.

Intuitively, a hybrid automaton is a "finite-state automaton" with continuous variables that evolve according to dynamics specified at each discrete *location* (or *mode*). The evolution of a hybrid automaton alternates *continuous* and *discrete* steps. In a continuous step, the location (i.e., the discrete state) does not change, while the continuous variables change following the continuous dynamics of the location. A discrete evolution step consists of the activation of a discrete transition that can change both the current location and the value of the state variables, in accordance with the reset function associated to the transition. The interleaving of continuous and discrete evolutions is decided by the invariant of the location, which must be true for the continuous evolution to proceed, and by the guard predicate of the transition, which must be true for a discrete transition to be active.

For example, the hybrid automaton in Fig. 5 can be used to specify the required behaviors of a triangle wave generator with period T and amplitude A. In the *Up* mode, the output y of the generator is required to increase with a constant slope until the internal variable t, initially set to zero, and increasing with a slope of one, reaches $T/2$. The generator will then switch to the *Down* mode, where y is required to decrease with the same slope, while t will keep on increasing until it crosses T. Once this threshold is crossed, the generator commutes back to the *Up* mode, while t is reset to zero.

Verifying safety of a hybrid automaton with respect to a prescribed set of bad states is equivalent to verifying that all legal behaviors of the automaton do not go through any of the bad states, i.e., the bad states are *unreachable*. The computation of the *reachable set*, which consists of all the states that can be reached under the dynamical evolution starting from a given initial state set, is nontrivial for hybrid automata. Since the states of a hybrid automaton are pairs made by a discrete location together with a vector of continuous variables, they have the cardinality of continuum. Therefore, in general, it is not possible to perform exact reachability analysis.

Hybrid automata come in several flavors. The original model allows for arbitrarily complex dynamics and was developed primarily for algorithmic analysis of hybrid systems [33]. The class of *hybrid input/output automata* enables compositional analysis of systems [38]. In *timed automata* [39], all the continuous variables are *clocks* (they have derivative 1) that can only be reset to zero. Many verification problems are decidable for this class, making it an interesting formalism for verification and requirement analysis. *Rectangular automata* [40] extend timed automata by allowing piecewise constant dynamics, while still keeping decidability of the reachability problem. *Linear hybrid automata* [41] extend rectangular automata by allowing guards and resets to be general linear predicates, at the price of losing decidability.

Hybrid Automata and Contracts: We can express contracts with hybrid automata by following the approach proposed by Benvenuti *et al.* [42]. We model the assumptions A with a hybrid automaton that generates all the admissible input sequences for a component (*uniform assumptions*), while we model the guarantees G as the set of admissible output sequences for the component. Then, a component M satisfies the contract if the behaviors of the composition of the hybrid automata for A and M are contained in G. When the guarantees are limited to *safety guarantees* ("nothing bad can happen"), then the contract satisfaction problem can be reduced to *reachability analysis* of a composition of automata.

Composition of contracts can be represented by appropriate composition operators on automata. For instance, the conjunction of assumptions corresponds to intersection of the associated automata, while their disjunction can be expressed by nondeterministic choice.

Under suitable restrictions, the other operations on contracts, as defined in Section II-C, can also be reduced to special instances of the reachability problem for timed or hybrid automata. Indeed, compatibility and consistency can be solved by checking whether the set of behaviors of the automaton describing, respectively, A and G is *empty*.

Checking refinement between two contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ is more involved. For uniform assumptions and safety guarantees [42], it is possible to associate to each contract the automaton $\mathcal{H}_A \parallel \mathcal{H}_G$, obtained by composition ($\parallel$) of the two automata $\mathcal{H}_A$ and $\mathcal{H}_G$, respectively describing the contract assumptions and guarantees. $\mathcal{H}_A \parallel \mathcal{H}_G$ models the behaviors admitted by the contract in the context of its legal environments. Then, if $A_2 \subseteq A_1$, contract refinement can be verified by checking the inclusion of the reachable sets of the two hybrid automata $\mathcal{H}_{A_1} \parallel \mathcal{H}_{G_1}$ and $\mathcal{H}_{A_2} \parallel \mathcal{H}_{G_2}$ associated with the contracts. When the evolution of the two hybrid automata cannot be computed exactly, this becomes a difficult task since it requires computing both overapproximations and underapproximations of the evolution, a capability supported by very few tools.

C. Verification Tools

As shown in Section III-A and B, the operations and relations on temporal logic and hybrid automata contracts can be reduced to basic verification tasks. In this section, we discuss some of the approaches reported in the literature to perform these tasks, together with the tools embodying them. Specifically, we focus on formal verification of hybrid models, which generates, in general, intractable problems, and classify the verification tools into five categories, based on the strategies adopted to deal with intractability.

1) Tools Based on Exact Reachability Set Computation: When the system dynamics are simple enough to be captured by timed or rectangular automata, their evolution can be computed exactly, and most of the verification techniques for finite-state models can be used to obtain an exact answer to verification problems.

A seminal tool in this category is KRONOS [43], which verifies real-time systems modeled by timed automata with respect to requirements expressed in the real-time logic Timed Computation Tree Logic (TCTL), using a backward–forward analysis approach.

The same approach was then extended to support rectangular automata in HyTech [44] by dealing with polyhedral state sets. A key feature of HyTech is its ability to perform parametric analysis, that is, to determine the values of design parameters for which a rectangular hybrid automaton satisfies a temporal-logic requirement. It can then be used as an evaluation engine for optimization-based design exploration, as discussed in Section V.

Modern tools use a different approach, based on an on-the-fly verification algorithm that does not need to build the entire reached set of the system. The most relevant tool using this approach is UPPAAL [45], written in Java and C++, and equipped with a graphical user interface. It handles real-time systems modeled as networks of timed automata, and complex properties expressed in a subset of CTL. Since the dynamics are represented just by clocks, it can support models with up to 100 of them. A comparison of the performance of the three tools above on the well-known railroad crossing example can be found in the literature [46].

2) Tools Based on Reachable Set Approximations: When the dynamics are more complex, the reachable set cannot be computed exactly. Nevertheless, approximation techniques can be used to obtain an answer in some cases. This approach is mainly used to verify *safety properties*. The system is safe if the reachable set is included in the safe set of states. Hence, *overapproximations* may be used to obtain positive answers, while *underapproximations* give negative answers.

One of the first tools that enabled verification of hybrid systems with complex dynamics is *d/dt* [47]. The tool approximates reachable states for hybrid automata where the continuous dynamics is defined by linear differential equations. Being one of the first approaches, the tool does not allow the composition of automata and is limited in scalability.

PHAVer [48] handles affine dynamics and guards and supports the composition of hybrid automata. The state space is represented using polytopes. Results are formally sound because of the exact and robust arithmetic with unlimited precision. Scalability is, however, limited. Models with more than 10 continuous variables are usually out of the capabilities of the tool.

SpaceEx [49] improves upon PHAVer in terms of scalability. Models with 100 variables have been analyzed with this tool. It combines polyhedra and support functions to represent the state space of systems with piecewise affine, nondeterministic dynamics. Differently from PHAVer, the result of SpaceEx is not guaranteed to be numerically sound. This means that when the tool states that the system is safe, we can only conclude that more sophisticated methods are necessary to find bugs for that system.

Flow* [50] supports systems with nonlinear ordinary differential equations (ODEs) (polynomial dynamics inside modes, polyhedral guards on discrete transitions) by representing the state space using Taylor models (bounded degree polynomials over the initial conditions and time, bloated by an interval). Results are guaranteed to be numerically sound but scalability is limited to a dozen variables.

Ariadne [42], [51] uses numerical methods based on the theory of computable analysis to manipulate real numbers, functions, and sets in the Euclidean space in order to verify hybrid systems with nonlinear dynamics, guards, and reset functions. It supports composition to build complex systems from simpler components and can compute both upper-approximations and lower-approximations of the reachable set, which play the role of over and under approximations. By combining them, Ariadne can provide both positive and negative answers to the verification of safety properties and other more complex problems. Its expressivity, however, affects performance and scalability, which is currently limited to models with up to 10 continuous variables.

An alternative approach to approximate the reachable set of a hybrid automaton is to drop the standard infinite precision semantics and adopt an ϵ-semantics where states whose distance is less than a fixed ϵ are indistinguishable. Under this assumption, the reachability problem for hybrid automata becomes decidable [52]. pyHybridAnalysis [53] is a Python package that implements the ϵ-semantics approach to symbolically compute an approximation of the reachability region of hybrid automata with semi-algebraic dynamics.

3) Tools Based on Discrete Abstractions: In this setting, the hybrid model under verification is first abstracted by a finite-state discrete model that approximates the original one. If the abstraction is not accurate enough to obtain an answer to the verification problem, it is improved until either an answer is found or the maximum number of refinement steps is reached [54], [55]. The main advantage of this approach is that, in some cases, an answer to the verification problem can be obtained with few refinement steps, even for very complex models.

The refinement algorithm proposed by Clarke *et al.* [55] has been implemented by CheckMate [56], a Matlab/Simulink toolbox for the simulation and verification of hybrid systems with linear and affine dynamics. The abstraction of the system is obtained with a method called flow pipe approximation, where the reachable set over a bounded time interval $[0, t]$ is approximated by the union of a sequence of convex polyhedra.

One of the first tools to extend this approach to nonlinear systems is HSOLVER [57], which uses constraint propagation and abstraction-refinement techniques to discretize the state space of the system and verify safety properties. HSOLVER supports systems with complex nonlinear dynamics and guards, but it does not support the composition of automata. Because of the particular state-space representation, it cannot provide a graphical output of the reachable set, but only a safe/possibly-unsafe answer to the verification problem.

HybridSAL [58] uses predicate abstraction to abstract the discrete dynamics and qualitative reasoning to abstract the continuous dynamics of polynomial hybrid systems. The algorithm can be applied compositionally to abstract a system described as a composition of automata. Results are guaranteed to be sound. Its scalability is limited: Only 10 continuous variables can be handled.

HyCOMP [59] uses a different approach, where the system is abstracted with a discrete but infinite-state model using an SMT approach. The abstraction is precise for piecewise constant dynamics and is an overapproximation for affine dynamics. Results are guaranteed to be sound (the SMT-solver uses infinite-precision arithmetic). The tool was tested successfully on models with 60 continuous variables with piecewise constant dynamics and 150 Boolean variables.

4) Tools Based on Automated Theorem Proving: Given a sufficiently expressive logic, the verification problem can be reduced to test whether a formula of the form $Sys \rightarrow Prop$ is *valid* (a logical tautology), where *Sys* is a representation of the system under verification and *Prop* is the property of interest. Automated theorem proving techniques can thus be used to solve the problem. While in principle this approach can easily manage parametric and partially specified systems, and properties of arbitrary complexity, very few tools exploit it in the context of hybrid systems. This is mainly due to the need for a complex temporal logic to describe the system in detail, and to the fact that automated theorem provers usually need some intervention from the user to guide the proof search and find an answer.

A tool using theorem proving techniques in the context of hybrid systems is KeyMaera [36], which combines deductive, real algebraic, and computer algebraic prover technologies. Systems and properties are specified using the temporal logic $d\mathcal{L}$. To automate the verification process, KeyMaera implements automatic proof strategies that decompose the hybrid system specification symbolically. The tool is particularly suitable for verifying parametric hybrid systems and has been used successfully for verifying collision avoidance in case studies from train control to air traffic management.

5) Tools Based on Simulation: A simulation-based approach can be used to verify black-box models (when the internal dynamics is unknown) or models of more complex systems, since simulation can be made more computationally feasible. However, simulation is simply a virtual test bench that gives answers as good as the questions that are asked, hence there is no guarantee that the system behaves correctly under all conditions. Simulation-based verification explores the state space of the system by computing a set of trajectories while hoping to cover as much as possible the relevant parts of the state space. If one of the trajectories violates the property, a *counterexample* is found and a negative answer to the verification problem is given. Otherwise, no conclusion can be made on the truth of the property since simulation cannot cover the entire state space. Similarly, simulation-based verification cannot be used, in general, to certify the satisfaction of a contract, but rather to monitor and detect possible violations.

A first tool based on simulation is Breach [60], a Matlab/C++ toolbox for the simulation, verification of temporal logic properties, and reachability analysis of dynamical systems, defined as systems of ODEs or by external modeling tools such as Simulink. It uses systematic simulation to compute an underapproximation of the reachable set based only on a finite (though possibly large) number of simulations. It supports complex properties in STL and parameter synthesis.

S-TaLiRo [61] is also a suite of tools for the analysis of continuous and hybrid dynamical systems using linear time temporal logic. Distributed as a Matlab toolbox, it uses a robustness metric to guide the state space exploration, exploiting randomized testing and stochastic optimization techniques to maximize the chance of finding a counterexample. Similarly to Breach, it supports complex properties in Metric Temporal Logic and parametric systems.

Finally, System Level Formal Verification (SLFV) [62] can prove system correctness notwithstanding uncontrollable events (such as faults, variation in system parameters, external disturbances) by exhaustively considering all the relevant simulation scenarios.

IV. PLATFORM COMPONENT-LIBRARY DEVELOPMENT

In the bottom–up phase of the design process, a library of components, models, and related contracts is developed for the plant and the embedded system. As shown in Fig. 4(a), components and contracts are *hierarchically organized* to represent the system at different levels of abstraction, e.g., steady-state, discrete-event, and hybrid levels. Typically, at the highest levels of abstraction, a *signal-flow* approach is more appropriate to CPS modeling, as is the case in signal processing, feedback control based on sensor outputs and actuator inputs, and in systems composed of unilateral devices [63]. In these cases, relations between system variables are better viewed in terms of inputs and outputs, and interconnections in terms of output-to-input assignments. Inputs are used to capture the influence of the environment on the system, while outputs are used to capture the influence of the system on the environment. At the lowest levels of abstraction, *acausal* models, without *a priori* distinction between inputs and outputs, may be more suitable to model the majority of physical (e.g., mechanical, electrical, hydraulic or thermal) components, which are generally governed by laws that merely impose relations (rather than functions) among system variables, and where interconnections mean that variables are shared (rather than assigned) among subsystems.

Reflecting the taxonomy of requirements, the component library is also *viewpoint and domain dependent*, following a similar approach as in the "rich component" libraries that were first proposed for automotive embedded systems [12]. At each level of abstraction, components are capable of exposing multiple, complementary viewpoints, associated with different design concerns and different formalisms (e.g., graphs, linear temporal logic, algebraic differential equations). Moreover, following the platform component definition in Section II-C, models include extra-functional (performance) metrics, such as timing, energy, and cost, in addition to the description of their behaviors.

Components and contracts can then be expressed using the same formalisms introduced in Section III, in the context of requirement analysis and system verification. A major challenge in this multiview and hierarchical modeling scenario remains to maintain consistency among models and views, often developed using domain-specific languages and tools, as the library evolves [3]. In this respect, the algebra of contracts can offer an effective way to incrementally check consistency or refinement among models. This information can then be stored in the library to speed up verification tasks at design time [15]. Moreover, vertical contracts can be used to establish conditions for an abstract, approximate model, to be a sound representation of a concrete model, i.e., to define when a model still retains enough precision to address specific design concerns, in spite of the vagueness required to make it manageable by analysis tools [5]. In the following, we briefly review the main languages and tools for system modeling and simulation, as well as a few attempts at their integration.

A. Languages and Tools for System Modeling and Simulation

A number of *modeling and interchange languages* have been proposed over the years to enable checking system properties, exploring alternative architectural solutions for the same set of requirements, and exchanging the system descriptions between the different tasks of the design flow (e.g., controller design, validation, verification, testing, and code generation). An exhaustive survey is out of the scope of this paper. Among the several languages and tools, we recall here:

- generic modeling and simulation frameworks, such as Matlab/Simulink[6] and Ptolemy II[7];
- hardware description languages, such as Verilog[8] and VHDL,[9] or transaction-level modeling languages, such as SystemC,[10] together with their respective analog-mixed-signal extensions[11];
- modeling languages specifically tailored for acausal multiphysics systems, such as Modelica,[12] supported by tools such as Dymola[13] or JModelica[14];
- languages for architecture modeling, such as the Systems Modeling Language (SysML)[15] and the Architecture Analysis and Design Language (AADL).[16]

While some of these languages and tools mostly focus on simulation, some others are also geared towards modeling, analysis, and verification of extra-functional properties.

A number of proposals have also appeared towards modeling languages specifically tailored to CPSs. One of the first examples of these languages is Charon [64]. Charon supports the hierarchical description of system architectures via the operations of instantiation, hiding, and parallel composition. Continuous behaviors can be specified using differential as well as algebraic constraints, all of which can be declared at various levels of the hierarchy. A few years later, Giotto [65] provided an abstract programming model for the implementation of embedded control systems with real-time constraints. Giotto allows the designer to specify time-triggered sensor readings, task invocations, actuator updates, and mode switches in a way that is independent from the implementation details. The code can then be annotated with platform-dependent constraints to automatize the validation of the model and the synthesis of the control software. A more recent modeling language proposal is the Hierarchical Timing Language (HTL) [66]. In HTL, critical timing constraints are specified within the language and forced by the compiler. Programs in HTL are extensible by adding new program modules and by refining individual program tasks. This mechanism is invariant under parallel composition and allows individual tasks to be implemented using external languages to ease interoperability.

All the above languages are not intended to be interchange formats, in that they generally lack the capability to easily interface with other tools. A first proposal for a truly platform-independent interchange format based on hybrid automata is the Hybrid System Interchange Format (HSIF) [67]. HSIF can represent networks of hybrid automata, albeit without hierarchy or modules. Variables can be shared or local, and the communication mechanism is based on broadcasting of Boolean signals. Other examples are the Metropolis meta-model [68], which also accounts for implementation considerations, such as equation sorting and event detection, and the interchange format for switched linear systems defined by Di Cairano *et al.* [69]. More recently, the Compositional Interchange Format (CIF) has been proposed to overcome some of the limitations of previous languages [70], such as the absence of hierarchy in HSIF, and the limitation to linear dynamics only [69]. CIF is a generic exchange format, integrating compositional semantics with automata, process communication, and synchronization based on shared events, differential algebraic equations, different forms of urgency, and process definition and instantiation to support reuse and large-scale system modeling. It can interface with a number of other languages and tools (e.g., Uppaal, PhaVer, Ariadne, Modelica, Matlab) and is currently used in both academia and industry.

As an alternative approach to facilitate the integration of different domains and models within a unifying framework, Shah *et al.* [71] propose the customization of SysML [72] by using profiles and domain-specific languages to support multiple representations (or architectures) of the system, and graph transformations to describe the relations between them.

Finally, particularly appealing for CPS modeling and simulation is the Functional Mockup Interface (FMI), an evolving standard for composing component models, which are better realized and characterized using distinct modeling tools [73], [74]. Initially developed within the MODELISAR project, and currently supported by a number of industrial partners and tools,[17] FMI shows promise for enabling the exchange and interoperation of model components. The FMI standard supports both co-simulation, where a component called Functional Mock-up Unit (FMU) implements its own simulation algorithm,

[6]http://www.mathworks.com/products/simulink
[7]http://ptolemy.eecs.berkeley.edu
[8]http://www.verilog.com/
[9]http://www.vhdl.org
[10]http://www.accellera.org/downloads/standards/systemc
[11]http://www.eda.org/verilog-ams/, http://www.eda.org/vhdl-ams/, http://www.accellera.org/downloads/standards/systemc/ams
[12]https://www.modelica.org/
[13]http://www.dynasim.se/
[14]http://www.jmodelica.org/
[15]SysML is an object-oriented modeling language largely based on the Unified Modeling Language (UML) 2.1, which also provides useful extensions for systems engineering (http://www.omg.org/spec/SysML).
[16]http://www.aadl.info/aadl/currentsite
[17]https://www.fmi-standard.org/

and model exchange, where an FMU exports sufficient information for an external simulation algorithm to execute simulation. However, while in principle FMI is capable of composing components representing timed behaviors, including physical dynamics and discrete events, several aspects of the standard—e.g., to guarantee that a composite model does not exhibit nondeterministic and unexpected behaviors—are currently object of investigation [75].

V. MAPPING SPECIFICATIONS TO IMPLEMENTATIONS

In the absence of a unified framework for automated synthesis of CPSs simultaneously subject to a heterogeneous set of requirements, we reason about different aspects or representations of the design by using specialized analysis and synthesis (mapping) frameworks that can operate with different formalisms. During design space exploration, both horizontal and vertical contracts can be used to define both the specification and the implementation platforms, thus playing an essential role in checking or enforcing that an aggregation of components is compatible, and that the implementation is a correct refinement of the specification.

At each abstraction level, *mapping* to a lower level can be performed by either leveraging a *synthesis* tool, or by solving an *optimization* problem that uses constraints from both the specification and the implementation layers to evaluate global tradeoffs among components. Accordingly, we denote as $\mathcal{C}_{\text{syn}}$ a contract that can be used as input of a specialized synthesis tool, and as $\mathcal{C}_{\text{opt}}$ a contract that serves as a conjunction of constraints in a more generic optimization problem. $\mathcal{C}_{\text{opt}}$ can be further characterized as $\mathcal{C}_{\text{ver}} \wedge \mathcal{C}_{\text{sim}}$, where $\mathcal{C}_{\text{ver}}$ denotes a contract whose satisfaction can be formally verified, e.g., using the tools introduced in Section III, while $\mathcal{C}_{\text{sim}}$ refers to a contract that can only be checked by simulation. In the following, we provide examples of mapping techniques and tools for the different design tasks in our methodology.

A. Architecture Design

In the design of the system architecture, $\mathcal{C}_{A,\text{syn}}$ in Fig. 4(a) includes the specification contract, e.g., expressed in terms of linear (or quadratic) arithmetic constraints on Boolean and real variables, as well as the steady-state models of the architecture, e.g., represented as constraints on a graph. Then, an implementation can be directly synthesized by solving a *mixed integer-linear (or quadratic) program* to minimize a cost function (e.g., component number, weight, cost, energy) while satisfying the constraints above [14]. It has been shown that the formulation above can encompass a variety of requirements, such as connectivity, safety, reliability, and energy balance [14]. These requirements are mapped on a representation of the system architecture, e.g., in terms of a labeled graph, where nodes represent the (parameterized) components and edges represent their interconnections.

To handle reliability requirements, the ArchEx framework [14], [76] implements two algorithms to decrease the complexity of exhaustively enumerating all failure cases on all possible graph configurations, namely, Integer-Linear Programming Modulo Reliability (ILP-MR) and Integer-Linear Programming with Approximate Reliability (ILP-AR). ILP-MR "lazily" combines an ILP solver with a background exact reliability analysis routine, inspired by similar approaches previously reported in the literature [29], [77]. The solver iteratively provides candidate configurations that are analyzed and accordingly modified to satisfy the reliability requirements. Although exact reliability analysis is an NP-hard problem, the idea is to perform it only when needed, i.e., a small number of times, and possibly on smaller graph instances. Conversely, ILP-AR efficiently generates a monolithic problem instance using an "eager" approach by leveraging approximate reliability computations that can still provide estimates to the correct order of magnitude and with an explicit theoretical bound on the approximation error. The synthesized architecture can then serve as a specification for the control design step.

B. Control Synthesis

Control synthesis deals with the problem of mapping (synthesizing) high-level formal requirements [e.g., $\mathcal{C}_{C,\text{syn}}$ in Fig. 4(a)] and a description of the plant into a lower-level, correct-by-construction controller that implements the desired requirements once it is composed with the plant. We review below the main techniques for the synthesis of control algorithms for CPSs.

1) Reactive Synthesis: When requirements are expressed using a discrete-time temporal logic (e.g., LTL or CTL), controller synthesis can be solved using techniques from *reactive synthesis*, which has been an active area of research since the late 1980s, and it is still attracting a considerable attention today [78]–[81]. In this case, the specifications are mapped on a DE implementation of the controller, e.g., in terms of a state machine that represents a lower level of abstraction in the design refinement process.

Let E and D be sets of environment (input) and controlled (output) variables, respectively, of a DE controller. Let $s = (e, d) \in \mathcal{E} \times \mathcal{D}$ be its state, and $\mathcal{C}_{\text{LTL}}$ an LTL contract of the form $(E \cup D, \varphi_e, \varphi_e \rightarrow \varphi_s)$, where φ_e characterizes the assumptions on the environment and φ_s characterizes the system requirements. Reactive synthesis can then be viewed as a two-player game between an environment that attempts to falsify the specification in $\mathcal{C}_{\text{LTL}}$ and a controlled plant that tries to satisfy it. A control strategy is a partial function $f : (s_0 s_1 \ldots s_{t-1}, e_t) \mapsto d_t$, which selects the value of the controlled variables based on the state sequence so far and the behavior of the environment so that the (controlled) system satisfies φ_s as long as the environment satisfies φ_e. If such a strategy exists, the specification is said to be *realizable*. For general LTL, the synthesis problem has a doubly exponential complexity. However, a subset of LTL, namely generalized reactivity (1) (GR(1)), generates problems that are polynomial in $|\mathcal{E} \times \mathcal{D}|$, the number of valuations of the variables in E and D [78]. Given a GR(1) specification, there are game solvers and digital design synthesis tools that generate a finite-state automaton that represents the control strategy for the system [81]–[85].

When the requirements also involve continuous variables, by "replacing" continuous dynamics by discrete abstractions, it is possible to reduce the synthesis problem to a purely discrete one and therefore within the realm of reactive synthesis or other established DE system control synthesis methods [86], [87], as available for instance in the third revision of the CIF language for supervisory control synthesis [88]. More recently, a synthesis method for discrete-time CPSs has been proposed based on a model predictive control framework [89], [90]. In [90] STL specifications are encoded as mixed integer-linear constraints on the system variables of an optimization problem that is solved at each step, following a receding horizon approach.

2) Synthesis by Abstraction: Because of the limited applicability of existing tools to large-scale CPS hybrid models, constructing effective abstractions in a compositional way is key in order to tackle the synthesis problem. Indeed, the notion of approximate bisimulation [91] has been recently introduced to obtain correct and complete abstractions of differential equations that can be used to solve controller design problems. PESSOA [92] is a software toolbox, which exploits approximate bisimulation to implement efficient synthesis algorithms operating over the equivalent finite-state machine models. The resulting controllers are also finite-state and can be readily transformed into code for any desired digital platform. This transformation assigns the finite-state controller operation to a processor, where code is the result of mapping the controller equations into the instruction set of the processor.

Another approach to mapping a controller into a processor is the control software synthesis tool QKS [93]. Given the sampling time of the controller and the precision of the analog-to-digital conversion of state measurements, QKS can compute both the controllable region and an implementation in C code of a controller driving the system into a goal region in finite time.

A library-based compositional synthesis approach that directly conforms to the PBD paradigm has recently been presented to solve high-level motion planning problems for multirobot systems [94]. The desired behavior of a group of robots is specified using a set of safe LTL properties (top–down step of the flow). The closed-loop behavior of the robots under the action of different lower-level controllers is abstracted using a library of motion primitives, each of which corresponds to a controller that ensures a particular trajectory in a given configuration (bottom–up step of the flow). By relying on these primitives, the mapping problem is then encoded as an SMT problem and solved by using an off-the-shelf SMT solver to efficiently generate control strategies for the robots.

3) Hybrid Controller Synthesis: Several real-time constraints, mostly related to the physical plant and the hardware implementation of the controller, may require the full expressiveness of continuous and hybrid models. However, solving the controller synthesis problem by directly mapping to these abstractions is a very difficult task [95]. Even in the context of timed automata, where the synthesis problem is known to be solvable in an exact way [96], efficient and practical tools are lacking. One of the few exceptions is UPPAAL-TIGA [97], [98], an extension of UPPAAL that implements on-the-fly algorithms for solving the controller synthesis problem on timed automata with respect to reachability and safety properties expressed using timed computation tree logic.

Most of the algorithms for controller synthesis of hybrid automata subject to a safety specification are based on solving a differential game in which the environment is trying to drive the system into its target set at the same time as avoiding the target set of the controller. A general formulation for this problem can be found in the literature [99], [100]. Examples include the symbolic semi-algorithm to compute the controllable region of a linear hybrid automaton with respect to a safety goal [101] and a procedure to synthesize the maximal safe controller for more general hybrid systems with a lower bound on event separation [100]. One of the few publicly available tools implementing this two-person game approach is PHAVER+ [102], an extension of PHAVER that can automatically synthesize discrete controllers for linear hybrid automata with respect to safety and reachability goals.

Two synthesis (mapping) approaches have also been presented that can incorporate finite-precision sensors and actuators as well as the finite response time of the controller [103], [104]. In these works, the synthesis problem is addressed for two subclasses of hybrid automata, namely *elastic controllers*, and *lazy linear hybrid automata*, operating in an environment represented by hybrid automata. Elastic controllers are timed automata without invariants and with closed guards [105], [106]. They were introduced together with a parametric semantics for timed controllers called the Almost ASAP semantics, which relaxes the standard idealized As Soon As Possible (ASAP) semantics that cannot be implemented by any physical device no matter how fast it is. The result is that any correct Almost ASAP controller can be implemented by a program on a hardware if this hardware is fast enough. The first paper [103] presents a corresponding automated tool chain that can extract from an elastic controller a correct-by-construction HW/SW implementation described in SystemC. On the other hand, lazy linear hybrid automata [107] are used to model the discrete-time behavior of control systems containing finite-precision sensors and actuators interacting with their environment under bounded delays. A methodology and a corresponding tool chain to synthesize an implementable control strategy for LLHA are discussed in the second paper [104].

C. Optimized Mapping and Design Space Exploration

Whenever correct-by-construction synthesis from requirements results into intractable problems, it is still possible to cast the design exploration problem, in its more general terms, as an optimization problem, where the system specifications are checked by a formal verification engine or by monitoring simulation traces. For instance, let $\mathcal{C}_{\text{sim}} = (V, \phi_e, \phi_e \rightarrow \phi_s)$ be a contract that must be checked by simulation, where ϕ_e and ϕ_s are temporal logic formulas. Then, given an array of costs C, the mapping problem can be cast as a *multiobjective robust optimization* problem to find a set of configuration parameter vectors κ^* that are Pareto optimal with respect to the objectives in C, while guaranteeing that the system satisfies ϕ_s for all possible traces s satisfying the environment assumptions ϕ_e. More formally

$$\begin{aligned} &\min_{\kappa\in\mathcal{K},\pi\in\Pi} \quad C(\kappa,\pi) \\ &\text{s.t.} \begin{cases} \mathcal{F}(s,\kappa)=0 \\ s\models\phi_s(\pi) \qquad \forall s \text{ s.t. } s\models\phi_e(\pi) \end{cases} \end{aligned} \tag{18}$$

where π is a set of formula parameters used to capture degrees of freedom that are available in the system specifications, and whose final value can also be determined as a result of the optimization process. For a given parameter valuation κ, s_κ is shorthand notation for $s_\kappa(t) = \{u(t), y_\kappa(t), x_\kappa(t)\}$, the set of traces of input, output, and internal signals (which are also represented as sets of traces over time $t \in \mathbb{R}_{\geq 0}$) that are obtained by simulating the behavioral model $\mathcal{F}(.)$ defined in Section II-C. A multiobjective optimization algorithm with simulation in the loop can then be implemented to find the Pareto optimal solutions κ^*. While this may be expensive in general, it becomes the only affordable approach in many practical cases.

The mapping methodology above can also encompass contracts of the form $\mathcal{C}_{\text{ver}} = (V, \phi_e, \phi_e \rightarrow \phi_s)$ whose satisfaction can still be efficiently verified via formal methods, even if the synthesis problem is intractable. Moreover, it can be used to perform joint design exploration of the controller and its execution platform, while guaranteeing that their specifications, captured by vertical contracts, are consistent. As mentioned in Section II-C, the association of functionality to architectural services to evaluate the characteristics (such as latency, throughput, power, and energy) of a particular implementation by co-simulation of both a functional model and an architectural model of the system is supported by frameworks such as METRONOMY.

VI. AIRCRAFT POWER DISTRIBUTION DESIGN EXAMPLE

We illustrate the application of the methodology introduced in this paper to the design of supervisory control systems for aircraft power distribution [14], [108].

Fig. 6 shows a sample structure of an aircraft electric power system (EPS) in the form of a single-line diagram, a simplified notation for three-phase power systems [109]. *Generators* (e.g., two on the left and two on the right side of the aircraft, denoted as GEN in Fig. 6) deliver power to the loads (e.g., avionics, lighting, heating, and motors, not represented in Fig. 6) via high-voltage and low-voltage ac (HVAC, LVAC) and dc *buses* (HVDC, LVDC). *Auxiliary* power unit (APU) generators or *batteries* (Batt) are, instead, used when one of the primary generators fails. *Essential buses* (ESS in Fig. 6) supply loads that cannot be unpowered for more than a predefined period t_{max}, while nonessential buses supply loads that may be shed in the case of a fault. *Contactors* are electromechanical switches that are opened or closed to determine the power flow from sources to loads, and are shown as double bars in the figure. AC *transformers* (ACT) convert high-voltage to low-voltage ac power. *Rectifier* units (RUs) convert and route ac power to dc buses. Transformer rectifier units (TRUs) act both as transformers and rectifiers.

The goal of the *supervisory controller* (not represented in Fig. 6) is to react to changes in system conditions or failures and reroute power by appropriately actuating the contactors to ensure that essential buses are adequately powered. A pictorial representation of the proposed design flow as instantiated for the EPS is shown Fig. 4. In the following, we briefly summarize the main steps followed to map the top-level system requirements into a lower-level representation of both the plant architecture and the control algorithm, to be further refined during subsequent design steps. Our overview is based on the results reported by Nuzzo *et al.* [14].

A. Top-Level Requirements

As a first step, top-level requirements are captured in terms of a system contract $\mathcal{C}_S$ using an electric power system DSL, which enables automatic translation of the specifications from a set of predefined primitives to one (or more) of the back-end formalisms mentioned in Section III. The DSL can smoothly interface with preexisting tools, such as visual programs for single-line diagrams, typically used by system engineers. Representative examples of system assumptions (A) and guarantees (G) supported by the DSL are provided below.

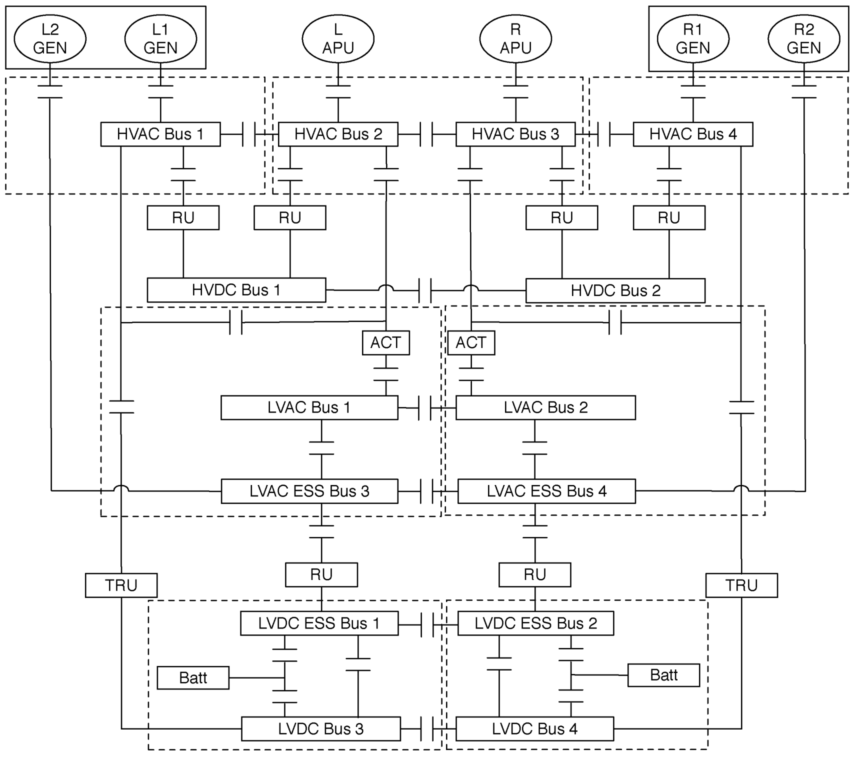


Fig. 6. *Single-line diagram of an aircraft electric power system (figure from [14]).*

A_1: **Reliability Level**. A typical power system specification would require that the failure probability for an essential bus (i.e., the probability of being unpowered for longer than $t_{\max}$ by any of the available generators) be smaller than a certain target r_S, e.g., corresponding to 10^{-9} per flight hour. We denote the probability r_S as the *reliability level* of the system. To allow formalizing this requirement, a set of environment assumptions must be provided to characterize the number and kind of component failures allowed, assuming that component failure events are all independent.

A_2: **Irreversible Failures**. As a second set of environment assumptions, we require that when a component fails during the flight, it will not come back online.

G_1: **Reliability Level**. As a first set of system-level guarantees, we need to ensure that the probability for an essential bus to be unpowered by any of the available generators r_T (i.e., the probability that there is no possible interconnection between the bus and any generator) is smaller than the required reliability level r_S, as defined above. We denote the probability r_T as the *topology reliability level*.

G_2: **Unhealthy Sources**. We require that the set of contactors directly connected to an unhealthy source be open to isolate it from the rest of the system.

G_3: **Operation in Nominal Conditions**. Under nominal conditions (i.e., when all generators and rectifier units are healthy), primary generators and rectifiers on each side of the electric power system topology must provide power to the buses on the same side; all other paths (and auxiliary power units) stay inactive.

G_4: **No Paralleling of AC Sources**. To avoid generator damage, ac sources should never be paralleled, i.e., no ac bus can be powered by multiple generators at the same time.

G_5: **System Reaction Time**. A dc essential bus can stay unpowered for no longer than $t_{\max}$ in case of failure.

The above system requirements are used to derive a contract $\mathcal{C}_T$ for the system architecture, in terms of arithmetic constraints on Boolean variables and failure probabilities (mixed integer-linear inequalities), and a contract $\mathcal{C}_C$ for the control algorithm, expressed as a conjunction of LTL and STL contracts, as shown in Fig. 4(b). Examples of DSL constructs supporting the above requirements and their translation into mixed integer-linear constraints and temporal logic formulas to capture different levels of abstraction can be found in the above-mentioned publication [14] and other related works [76], [108].

Architecture and control protocol need to be consistently designed to satisfy $\mathcal{C}_S$, which can be guaranteed by showing that $\mathcal{C}_T \otimes \mathcal{C}_C$ is compatible and $\mathcal{C}_T \otimes \mathcal{C}_C \preceq \mathcal{C}_S$. However, as discussed in Section II-C5, to enforce the correctness of the refinement between different levels of abstraction of the design platform, including different viewpoints, contract consistency and compatibility should hold in both the horizontal and vertical directions. While the composite contract $\mathcal{C}_T \otimes \mathcal{C}_C$ can be effectively used to model the *horizontal* interaction between the controller and its plant, to guarantee the overall system reliability and real-time performance, we also need to prove compatibility and consistency of the *vertical* contracts between architecture and control for the *timing* and *reliability* viewpoints. Specifically, the control protocol makes several assumptions that must be discharged by the architecture, e.g., in terms of the topology reliability level, due to the available component redundancy, and the worst-case latency, due to delays in both the physical plant and the execution platform.

While the proofs of compatibility and correctness of $\mathcal{C}_T \otimes \mathcal{C}_C$ are performed manually [14], reasoning with contracts is still instrumental to efficient co-design of architecture and control. In particular, Propositions 6.1 and 6.2 in the aforementioned paper [14] show that, if system-level requirements are "partitioned" according to $\mathcal{C}_T$ and $\mathcal{C}_C$, then the system can be designed in a compositional way, i.e., the architecture and control design steps summarized in Section VI-B and C can be *independently* refined while guaranteeing that the assembled system satisfies $\mathcal{C}_S$.

More specifically, given a system reliability requirement r_S, Proposition 6.2 states that, if the power system topology is synthesized to implement the contract $\mathcal{C}_T$ with a reliability level $r_T \leq r_S$, then there exists a time T^* (a function of the synthesized topology and the contactor actuation delays) such that a centralized controller implementing the contract $\mathcal{C}_C$ for the given topology with a reliability level r_S and $t_{\max} \geq T^*$ can also be synthesized, and the resulting controlled system is guaranteed to satisfy the top-level requirements.

B. Architecture Design

At the structural (steady-state) level of abstraction, the *plant architecture* is modeled as a directed graph, where each node represents a component (with the exception of contactors, which are associated with edges) and each edge represents an interconnection, oriented based on the direction of the power flow. The platform library $\mathcal{L}$ includes, as attributes, generator power ratings, component costs, and failure probabilities, in addition to a set of interconnection rules. The *supervisory controller* is abstracted as one or more finite state machines, which actuate the contactors in the plant to configure the network and route power from the generators to the loads based on the failure status of the components. The controller is characterized by a reaction time T_r.

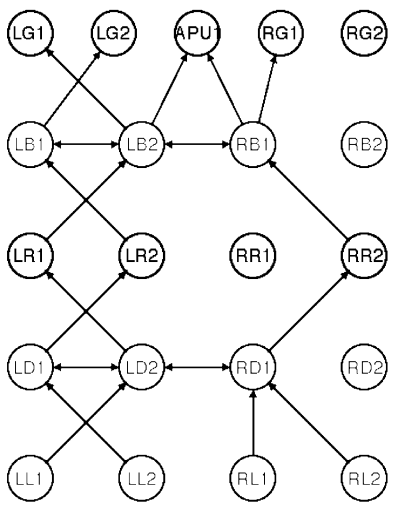
Based on the platform library described above, the contract $\mathcal{C}_T$, including the safety, connectivity, power flow, and reliability requirements for the system architecture, can be expressed (both assumptions and guarantees) in terms of linear inequalities on a set of Boolean variables, each denoting the presence or absence of an interconnection in the topology graph, as detailed in Section V. The

Generators	g (kW)
LG1	70
LG2	50
RG1	80
RG2	30
APU	100

Loads	l (kW)
LL1	30
LL2	10
RL1	10
RL2	20

Components	c
Generator	g/10
Bus	2000
Rectifier	2000
Contactor	1000

(a)



(b)

Fig. 7. ***Electric power system architecture design example. (a) Components and attributes used in the topology: generator power ratings, load power requirements, component costs. Generators, buses, and rectifiers are assumed to fail, during a mission, each with a probability of 2×10^{-4}. (b) Topology obtained after running ArchEx (ILP-MR algorithm) with a required reliability level $r_T^* \leq 2 \times 10^{-10}$. The topology diagram consists of rows of (from top to bottom) generators, ac buses, rectifier units, dc buses, and dc loads. Disconnected nodes are not used in the final topology. The resulting reliability level is $r_T = 0.79 \times 10^{-10}$ [76].***

tradeoff between redundancy and cost can then be explored using ArchEx to select an optimized system architecture, which is then offered as a specification for the control refinement step.

As represented in Fig. 4(c), the ILP-MR algorithm implemented in ArchEx, using Cplex [110] as a back-end optimization engine, is able to generate, in a few seconds, architectures for the primary distribution of an electric power system for different reliability requirements. Fig. 7(b) shows the architecture obtained after executing the ILP-MR algorithm using the parameters in Fig. 7(a), for a required reliability level $r_T^* \leq 2 \times 10^{-10}$. The resulting reliability level is $r_T = 0.79 \times 10^{-10}$, while the runtime is about 38 s on an Intel Core i7 2.8-GHz processor with 8-GB memory. Optimized architectures including up to 50 nodes can be selected in about 3 min [76].

C. Control Design

For a given architecture, the controller requirements can be defined as a contract $\mathcal{C}_C$, where the assumptions A_C encode the allowable behaviors of the environment (including the physical plant) and the guarantees G_C encode the desired behaviors of the closed-loop system, i.e., the guarantees of the system contract $\mathcal{C}_S$.

In this example, $\mathcal{C}_C$ can be expressed as the conjunction between an LTL contract $\mathcal{C}_{\text{LTL}}$ and an STL contract $\mathcal{C}_{\text{STL}}$. The STL formulas in $\mathcal{C}_{\text{STL}}$ can either be obtained by heterogeneous refinement of a subset of LTL formulas in $\mathcal{C}_{\mathrm{LTL}}$ or generated anew to capture design aspects related to the plant and the hardware implementation of the control algorithm, which cannot be expressed using the Boolean, untimed, or DE abstractions offered by LTL. $\mathcal{C}_{\mathrm{LTL}} \wedge \mathcal{C}_{\mathrm{STL}}$ is then a *vertical contract* for the controller since $\mathcal{C}_{\mathrm{LTL}}$ and $\mathcal{C}_{\mathrm{STL}}$ refer to two different controller representations, possibly involving different viewpoints (e.g., functional and timing).

To further illustrate this concept, we consider a simple power network topology consisting of a bus B connected with generators G_1 and G_2 via contactors C_1 and C_2, respectively. Moreover, we assume the following requirement for the controller: "If G_1 fails and G_2 is healthy, then the controller shall first open C_1 and then close C_2, while guaranteeing that B does not lose power for more than $t_{\max}$." We can encode this requirement as a conjunction of LTL and STL contracts as follows:

- Let g_i and c_i $(i = 1, 2)$ be Boolean variables encoding the status of generators (healthy/unhealthy) and contactors (open/closed). Then, the LTL contract can be used to capture the desired sequence of actions prescribed by the requirement, i.e., $\mathcal{C}_{\mathrm{LTL}} = (\{g_1, g_2, c_1, c_2\}, \mathtt{T}, \Box\{\neg g_1 \wedge g_2 \rightarrow \neg c_1 \wedge (\bigcirc c_2)\})$.
- Furthermore, we can refine the "absence of power losses on a bus for more than $t_{\max}$" with the statement: "the bus voltage V_B deviates from the desired value V_{d} by more than a margin ϵ for more than $t_{\max}$," and use the following STL formula to state that the above faulty behavior should never happen: $\phi = \neg(\diamond_{[0,\infty)} \Box_{[0,t_{\max}]}(|V_B(t) - V_d| \geq \epsilon))$. This translates into the contract $\mathcal{C}_{\mathrm{STL}} = (\{V_B\}, \mathtt{T}, \phi)$.

To guarantee the consistency of $\mathcal{C}_{\mathrm{LTL}} \wedge \mathcal{C}_{\mathrm{STL}}$ and refine it towards an implementation, the controller design process consists of two steps:

1) *Reactive Synthesis*. As shown in Fig. 4(d), $\mathcal{C}_{\mathrm{LTL}}$ is first used together with DE models of the plant components (also described by LTL formulas) to synthesize a reactive control protocol in the form of one (or more) state machines using reactive synthesis techniques, as described in Section V-B1. The resulting controller will satisfy $\mathcal{C}_{\mathrm{LTL}}$ by construction.
2) *Optimized Mapping*. The functional model of the synthesized controller is embedded into a high-fidelity hybrid model of the system, including an acausal representation of the plant. The entire system is simulated, under the assumption that the controller operates in a synchronous fashion, according to its reaction time T_r.[18] The satisfaction of $\mathcal{C}_{\mathrm{STL}}$ is then assessed by monitoring simulation traces, while optimizing a set of system parameters and costs, as described in Section V-C. The resulting optimal controller and plant configurations are returned as the final design, as shown in Fig. 4(g).

We observe that the joint execution of the controller with the plant in the mapping step effectively implements the synchronization mechanism that is instrumental in: 1) checking the consistency of the vertical contract; 2) discharging the timing assumptions made during the previous design steps; and 3) ultimately verifying the satisfaction of both the functional and timing viewpoints. However, mapping via simulation may be expensive to perform for certain kinds of requirements; reactive synthesis is then key to make it affordable by guaranteeing that several functional, safety, and reliability requirements are already satisfied by construction. In the following, we provide implementation details of the design steps above when applied to the system requirements in Section VI-A and the topologies described in Section VI-B.

1) Reactive Synthesis: Because the formulas in $\mathcal{C}_{\mathrm{LTL}}$ can be conveniently expressed using the GR(1) fragment of LTL, a control protocol can be automatically synthesized using the TuLiP Toolbox [81]. For different topologies, as the one shown in Fig. 7, and reliability levels $(r_S = r_T)$ ranging from 4×10^{-4} to 2.6×10^{-15}, a set of centralized and distributed control protocols were synthesized in approximately 0.5–2 s, with a number of states ranging from 4 to 113 [14], [108].

2) Optimized Mapping: A hybrid model implemented in Simulink, based on blocks from the SimPowerSystems library, as shown in Fig. 4(g), was used to analyze and optimize the closed-loop real-time performance of the controller, imported as a Matlab function. In this model, contactors have a nonideal response, affected by a fixed delay T_d on both the open and close commands. Therefore, it is possible to explore the tradeoff between controller reaction time T_r and contactor delay T_d and find, for instance, the maximum allowed reaction time T_r^* for a fixed T_d^*, in such a way that the essential dc bus is never out of range for more than $t_{\max}$. To do so, an optimization problem is cast following the formulation in (18). The constraints are expressed using STL formulas parametrized by T_r, and the system behavior is the set of traces $s = \{u, V_{\mathrm{DC}}\}$, where V_{DC} is the dc bus voltage signal to be observed during simulation and u spans the set of all admissible failure injection traces that are consistent with the assumptions on the reliability level in $\mathcal{C}_C$. The Breach toolbox [60] was used to monitor the satisfaction of STL formulas on the simulation traces.

As an example, for the architecture in Fig. 4(g), Fig. 4(e) shows the simulated voltage V_{B_3} of bus B_3 as a function of time for $T_r = 15$ ms, $T_d = 15$ ms, $t_{\max} = 70$ ms, and in the

[18]The synchronous operation of the controller is a design choice specific to this case study; it is not meant to serve as a general design guideline.

worst-case scenario of cascaded faults in generators G_1, G_2 and rectifier R_1 [108]. The red signal at the bottom of the figure is interpreted as a Boolean signal, which is high (one) when the requirement on the essential dc bus is violated, and low (zero) otherwise. The requirement on the DC bus is violated for 32 ms. Therefore, ($T_r = 15$ ms, $T_d = 15$ ms) is an unsafe parameter set.

The T_r versus T_d design space is explored in Fig. 4(f) by sampling the parameter space in approximately 4 h (on an Intel Core i7 2.3-GHz processor) to populate a 13×13 point grid [108]. The amount of elapsed time while the dc bus voltage is out of range, i.e., when the requirement on the dc bus is violated, is compared to the hard threshold $t_{\max} = 70$ ms, thus providing the designer with a "safe" region [marked in blue in Fig. 4(f)] for selecting the controller reaction time T_r as a function of the contactor delay. As an example, for $T_d = 20$ ms the maximum controller reaction time T_r^* allowed for safe operation and correct design is 4 ms.

VII. CONCLUSION

We presented a methodology that addresses the complexity and heterogeneity of cyber-physical systems by leveraging a contract framework to formalize the design process in a hierarchical and compositional way, and interconnect different modeling, analysis, and synthesis tools, to ensure quality and correctness of the final result. We surveyed formalisms and tools that can support the methodology at different levels of abstraction, from the level of discrete systems to the one of hybrid systems. To illustrate the application of the methodology, we used a concrete example of controller design in aircraft electric power systems.

The Way Forward: Extensions to the Methodology. Inspired by the design example, we envision a scenario in which a design management feature that we call a front-end *orchestrator* directly interacts with the designer, helps coordinate the set of back-end specialized tools, and consistently processes their results. For such an orchestrator to be developed, it is essential to develop *algorithms* that can maximally leverage the modularity offered by contracts by directly working on their representations to perform compatibility, consistency, and refinement checks on system portions of manageable size and complexity. Moreover, these algorithms should take advantage of any violation of the design constraints, i.e., a "counterexample" for system correctness, to provide meaningful *feedback* to the designer, and possibly set up *learning* strategies to refine or augment both the contract assumptions and guarantees until a final implementation is reached.

To better illustrate our methodology, we considered an abstract representation of a CPS in terms of composition between a controller and a plant. However, the concepts discussed in this paper are general enough to encompass several other, if not all, categories of CPSs. Specifically, because of the rigorous formalization of both the horizontal and vertical interactions between components, contracts seem to offer a "natural" theoretical framework for the design of provably correct *distributed* and *hierarchical control* systems in a scalable way. In this respect, to support the design of adaptive architectures, in which components (agents) can dynamically reconfigure themselves, e.g., by changing their locations or communication patterns, the challenge is to provide mechanisms that can efficiently export at the *architectural exploration level* the most important constraints and metrics imposed by the lower-level *system dynamics* and *network fabrics*. Accordingly, as an integral part of the *execution platform refinement* process, which was not covered in this paper, our framework can be extended to incorporate several design space exploration methodologies across the hardware, software, and communication layers, which are being consolidated over the years by the joint effort of both academia and industry. Arguably, we can further extend the design paradigm discussed in this paper to systems whose main objective is to sense and monitor a physical "plant" and process the collected data, rather than controlling it. In this case, contracts would rather be used to capture the interaction of the physical world with the *sensing, identification, data analysis, or learning algorithms*, and their deployment on the embedded platform.

Finally, we observe that several parameters impacting the behavior of CPSs are subject to variability due to manufacturing tolerances, usage, and faults. Moreover, the models that are normally used to design multiphysics systems inevitably introduce inaccuracies [25]. A survey on formalisms and tools for stochastic system design is out of the scope of this paper. However, the importance of providing a better support for reasoning about the probabilistic properties of systems and the deployment of robust design techniques cannot be overemphasized. In this context, advancing the state of the art in compositional approaches for *stochastic systems* and *stochastic contract frameworks* (e.g., see [111]–[113]) is deemed as essential to improve on the scalability of stochastic analysis and synthesis techniques (e.g., see [114] and [115]) and make their adoption actually feasible in current design flows. ■

REFERENCES

[1] J. Sztipanovits, "Composition of cyber-physical systems," in *Proc. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst.*, Mar. 2007, pp. 3–6.

[2] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. IEEE Int. Symp. Object Oriented Real-Time Distrib. Comput.*, May 2008, pp. 363–369.

[3] P. Nuzzo and A. Sangiovanni-Vincentelli, "Let's get physical: Computer science meets systems," in *From Programs to Systems. The Systems Perspective in Computing*, vol. 8415, S. Bensalem, Y. Lakhneck, and A. Legay, Eds. Berlin, Germany: Springer-Verlag, 2014, pp. 193–208.

[4] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proc. IEEE*, ser. 95, no. 3, pp. 467–506, Mar. 2007.

[5] P. Nuzzo, A. Sangiovanni-Vincentelli, X. Sun, and A. Puggelli, "Methodology for the design of analog integrated interfaces using contracts," *IEEE Sensors J.*, vol. 12, no. 12, pp. 3329–3345, Dec. 2012.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2008.

[7] A. Benveniste *et al.*, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects*. New York, NY, USA: Springer-Verlag, 2008, pp. 200–225.

[8] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. Symp. Found. Softw. Eng.*, 2001, pp. 109–120.

[9] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems," *Eur. J. Control*, vol. 18-3, no. 3, pp. 217–238, 2012.

[10] A. Benveniste *et al.*, "Contracts for system design," INRIA, Rapport de recherche RR-8147, Nov. 2012.

[11] M. Masin *et al.*, "META II: Lingua franca design and integration language," IBM Research, Tech. Rep, Aug. 2011. [Online]. Available: http://www.darpa.mil/uploadedFiles/Content/Our_Work/TTO/Programs/AVM/IBM META Final Report.pdf

[12] W. Damm *et al.*, "Boosting re-use of embedded automotive applications through rich components," in *Proc. Found. Interface Technol.*, 2005, pp. 1–18.

[13] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *Proc. Design, Autom. Test Eur.*, Mar. 2011, pp. 1–6.

[14] P. Nuzzo *et al.*, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[15] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *Proc. Design, Autom. Test Eur.*, Mar. 2014, pp. 1–6.

[16] M. Maasoumy, P. Nuzzo, and A. Sangiovanni-Vincentelli, "Smart buildings in the smart grid: Contract-based design of an integrated energy management system," in *Cyber Physical Systems Approach to Smart Electric Power Grid*, S. K. Khaitan, J. D. McCalley, and C. C. Liu, Eds. Berlin, Germany: Springer-Verlag, 2015, pp. 103–132.

[17] P. Nuzzo, A. L. Sangiovanni-Vincentelli, and R. M. Murray, "Methodology and tools for next generation cyber-physical systems: The iCyPhy approach," in *Proc. INCOSE Int. Symp.*, Jul. 2015.

[18] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *Proc. Int. Conf. Formal Methods Models Co-Design*, Oct. 2014, pp. 104–113, DOI: 10.1109/MEMCOD.2014.6961848.

[19] L. Benvenuti, A. Ferrari, E. Mazzi, and A. Sangiovanni-Vincentelli, "Contract-based design for computation and verification of a closed-loop hybrid system," in *Proc. Hybrid Syst.: Comput. Control*, 2008, pp. 58–71.

[20] S. Graf, R. Passerone, and S. Quinton, "Contract-based reasoning for component systems with rich interactions," in *Embedded Systems Development*, vol. 20, A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, Eds. New York, NY, USA: Springer-Verlag, 2014, pp. 139–154.

[21] R. Alur and T. Henzinger, "Reactive modules," *Formal Methods Syst. Design*, vol. 15, no. 1, pp. 7–48, 1999.

[22] F. Balarin *et al.*, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, no. 4, 2003.

[23] F. Balarin *et al.*, "Platform-based design and frameworks: Metropolis and metro ii," in *Model-Based Design for Embedded Systems*, G. Nicolescu and P. J. Mosterman, Eds. Boca Raton, FL, USA: CRC Press, Taylor & Francis, Nov. 2009, p. 259.

[24] L. Guo *et al.*, "Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems," in *Proc. Int. Conf. Hardw.-Softw. Codesign Syst. Synthesis*, Oct. 2014.

[25] P. Nuzzo and A. Sangiovanni-Vincentelli, "Robustness in analog systems: Design techniques, methodologies and tools," in *Proc. IEEE Symp. Ind. Embedded Syst.*, Jun. 2011, pp. 194–203.

[26] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, "Cyber-physical system design contracts," in *Proc. Int. Conf. Cyber-Phys. Syst.*, 2013, pp. 109–118.

[27] T. Bienmüller, W. Damm, and H. Wittke, "The Statemate verification environment," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 1855, E. A. Emerson and A. P. Sistla, Eds. Berlin, Germany: Springer-Verlag, 2000, pp. 561–567.

[28] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009.

[29] P. Nuzzo, A. Puggelli, S. Seshia, and A. Sangiovanni-Vincentelli, "CalCS: SMT solving for non-linear convex constraints," in *Proc. Formal Methods Comput.-Aided Design*, Oct. 2010, pp. 71–79.

[30] A. Pnueli, "The temporal logic of programs," in *Proc. Annu. Symp. Found. Comput. Sci.*, Nov. 1977, pp. 46–57.

[31] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. Formal Modeling Anal. Timed Syst.*, 2004, pp. 152–166.

[32] A. Cimatti, M. Roveri, and S. Tonetta, "Requirements validation for hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 5643, A. Bouajjani and O. Maler, Eds. Berlin, Germany: Springer-Verlag, 2009, pp. 188–203.

[33] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Syst.*, vol. 736, New York, NY, USA: Springer-Verlag, 1993, pp. 209–229.

[34] W. Damm, G. Pinto, and S. Ratschan, "Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems," *Int. J. Found. Comput. Sci.*, vol. 18, no. 1, pp. 63–86, 2007.

[35] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.

[36] A. Platzer, *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Heidelberg, Germany: Springer-Verlag, 2010.

[37] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *Int. J. Softw. Tools Technol. Transfer*, vol. 4, no. 2, pp. 224–233, 2003.

[38] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid I/O automata," *Inf. Comput.*, vol. 185, no. 1, pp. 105–157, 2003.

[39] R. Alur and D. L. Dill. "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8.

[40] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. "What's decidable about hybrid automata?," *J. Comput. Syst. Sci.*, vol. 57, no. 1, pp. 94–124, 1998.

[41] T. A. Henzinger, "The theory of hybrid automata," in *Proc. IEEE Symp. Logic Comput. Sci.*, Jul. 1996, pp. 278–292.

[42] L. Benvenuti *et al.*, "Assume-guarantee verification of nonlinear hybrid systems with ARIADNE," *Int. J. Robust Nonlinear Control*, vol. 24, no. 4, pp. 699–724, 2014.

[43] S. Yovine. "KRONOS: A verification tool for real-time systems," *Int. J. Softw. Tools Technol. Transfer*, vol. 1, no. 1–2, pp. 123–133, 1997.

[44] T. A. Henzinger, P. Ho, and H. Wong-Toi, "HYTECH: A model checker for hybrid systems," *Int. J. Softw. Tools Technol. Transfer*, vol. 1, no. 1–2, pp. 110–122, 1997.

[45] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi. "Developing UPPAAL over 15 years," *Softw., Pract. Exper.*, vol. 41, no. 2, pp. 133–142. , 2011.

[46] B. Bérard and L. Sierra, "Comparing verification with HyTech, KRONOS and Uppaal on the railroad crossing example," CNRS & ENS de Chachan, France, Tech. Rep. LSV-00-2, 2000.

[47] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Proc. Hybrid Systems: Computation and Control*, vol. 1790. LNCS, Berlin, Germany: Springer-Verlag, 2000, pp. 20–31.

[48] G. Frehse, "PHAVer: Algorithmic verification of hybrid systems past HyTech," *Int. J. Softw. Tools Technol. Transfer*, vol. 10, pp. 263–279, 2008.

[49] G. Frehse *et al.*, "SpaceEx: Scalable verification of hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*," LNCS, vol. 6806. Berlin, Germany: Springer, 2011, pp. 379–395.

[50] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*, Lecture Notes in Computer Science, vol. 8044. Berlin, Germany: Springer-Verlag, 2013 pp. 258–263.

[51] L. Benvenuti *et al.*, "Ariadne: Dominance checking of nonlinear hybrid automata using reachability analysis," in *Reachability Problems*, vol. 7550, A. Finkel, J. Leroux, and I. Potapov, Eds. Berlin, Germany: Springer-Verlag, 2012, pp. 79–91.

[52] A. Casagrande, C. Piazza, and A. Policriti. "Discrete semantics for hybrid automata," *Discrete Event Dynamic Syst.*, vol. 19, no. 4, pp. 471–493. 2009.

[53] A. Casagrande and T. Dreossi, "pyHybrid analysis: A package for semantics analysis of hybrid systems," in *Proc. Euromicro Conf. Digital Syst. Design*, Sep. 2013, pp. 815–818.

[54] R. Alur, T. Dang, and F. Ivančić, "Counterexample-guided predicate abstraction of hybrid systems," *Theor. Comput. Sci.*, vol. 354, no. 2, pp. 250–271, 2006.

[55] E. M. Clarke *et al.* (2003). "Abstraction and counterexample-guided refinement in model checking of hybrid systems," *Int. J. Found. Comput. Sci.*, vol. 14, no. 4, pp. 583–604, 2003.

[56] B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan, "Modeling and verification of hybrid dynamical system using CheckMate," in *Proc. Int. Conf. Autom. Mixed Processes: Hybrid Dynamic Syst.*, 2000, pp. 1–7.

[57] S. Ratschan and Z. She, "Safety verification of hybrid systems by constraint propagation based abstraction refinement," *Trans. Embedded Comput. Syst.*, vol. 6, no. 1, 2007, Art. ID. 8.

[58] A. Tiwari. "Abstractions for hybrid systems," *Formal Methods Syst. Design*, vol. 32, no. 1, pp. 57–83, 2008.

[59] A. Cimatti, S. Mover, and S. Tonetta. "SMT-based scenario verification for hybrid systems," *Formal Methods Syst. Design*, vol. 42, no. 1, pp. 46–66, 2013.

[60] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*. Berlin, Germany: Springer-Verlag, 2010, pp. 167–170.

[61] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *Proc. Int. Conf. Tools Algor. Construct. Anal. Syst.*, 2011, pp. 254–257.

[62] T. Mancini *et al.*, "System level formal verification via model checking driven simulation," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 8044, New York, NY, USA: Springer-Verlag, 2013, pp. 296–312.

[63] J. C. Willems, "The behavioral approach to open and interconnected systems," *IEEE Control Syst.*, vol. 27, no. 6, pp. 46–99, Dec. 2007.

[64] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee, "Modular specification of hybrid systems in Charon," in *Proc. Hybrid Systems: Computation and Control*, vol. 1790, New York, NY, USA: Springer, 2000, pp. 6–19.

[65] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proc. IEEE*, vol. 91, no. 1, pp. 84–99, Jan. 2003.

[66] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan, "A hierarchical coordination language for interacting real-time tasks," in *Proc. ACM IEEE Int. Conf. Embedded Softw.*, 2006, pp. 132–141.

[67] MoBIES team, HSIF semantics, University of Pennsylvania, Tech. Rep., 2002.

[68] A. Pinto, L. P. Carloni, R. Passerone, and A. L. Sangiovanni-Vincentelli, "Interchange format for hybrid systems: Abstract semantics," in *Proc. Hybrid Syst.: Comput. Control*, Mar. 2006, pp. 491–506.

[69] S. Di Cairano, A. Bemporad, M. Kvasnica, and M. Morari, "An architecture for data interchange of switched linear systems," HYCON Network of Excellence, Deliverable workpackage 3.D, 2006.

[70] D. E. N. Agut, D. A. van Beek, and J. E. Rooda, "Syntax and semantics of the compositional interchange format for hybrid systems," *J. Log. Algebr. Program.*, vol. 82, no. 1, pp. 1–52, 2013.

[71] A. A. Shah, D. Schaefer, and C. J. J. Paredis, "Enabling multi-view modeling with SysML profiles and model transformations," in *Proc. Int. Conf. Product Lifecycle Manage.*, 2009, pp. 527–538.

[72] OMG systems modeling language. [Online]. Available: http://www.sysml.org/.

[73] T. Blochwitz *et al.*, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proc. 9th Int. Modelica Conf.*, 2012, pp. 173–184, DOI: 10.3384/ecp12076173.

[74] MODELISAR Consortium and Modelica Association, Functional mock-up interface for co-simulation. Version 1.0, Oct. 2010. [Online]. Available: https://www.fmi-standard.org.

[75] D. Broman *et al.*, "Determinate composition of FMUs for co-simulation," in *Proc. ACM IEEE Int. Conf. Embedded Softw.*, 2013, pp. 2:1–2:12.

[76] N. Bajaj, P. Nuzzo, M. Masin, and A. L. Sangiovanni-Vincentelli, "Optimized selection of reliable and cost-effective cyber-physical system architectures," in *Proc. Design, Autom. Test Eur.*, Mar. 2015, pp. 561–566.

[77] C. Hang, P. Manolios, and V. Papavasileiou, "Synthesizing cyber-physical architectural models with real-time constraints," in *Proc. Int. Conf. Comput.-Aided Verification*, Dec. 2011, pp. 441–456.

[78] N. Piterman and A. Pnueli, "Synthesis of reactive(1) designs," in *Proc. Verification, Model Checking, Abstract Interpretation*, 2006, pp. 364–380.

[79] M. Kloetzer and C. Belta, "A fully automated framework for control of linear systems from temporal logic specifications," *IEEE Trans. Autom. Control*, vol. 53, no. 1, pp. 287–297, Feb. 2008.

[80] H. Kress-Gazit, G. Fainekos, and G. Pappas, "Temporal-logic-based reactive mission and motion planning," *IEEE Trans. Robot.*, vol. 25, no. 6, pp. 1370–1381, Dec. 2009.

[81] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: A software toolbox for receding horizon temporal logic planning," in *Proc. Int. Conf. Hybrid Syst.: Comput. Control*, 2011, pp. 313–314.

[82] A. Pnueli, Y. Saar, and L. D. Zuck, "JTLV: A framework for developing verification algorithms," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 6174, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Germany: Springer, 2010, pp. 171–174.

[83] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 4590, W. Damm and H. Hermanns, Eds. Berlin, Germany: Springer, 2007, pp. 258–262.

[84] R. Bloem *et al.*, "RATSY—A new requirements analysis tool with synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 6174, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Germany: Springer, 2010, pp. 425–429.

[85] R. Bloem *et al.* (2014). "Synthesizing robust systems," *Acta Informatica*, vol. 51, no. 3–4, pp. 193–220, 2014.

[86] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[87] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. SpringerLink Engineering, New York, NY, USA: Springer-Verlag, 2008.

[88] D. A. van Beek *et al.*, "CIF 3: Model-based engineering of supervisory controllers," in *Proc. Int. Conf. Tools Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. New York, NY, USA: Springer-Verlag, 2014, pp. 575–580.

[89] M. Maasoumy *et al.*, "Optimal load management system for aircraft electric power distribution," in *Proc. Int. Conf. Decision Control*, Dec. 2013, pp. 2939–2945.

[90] V. Raman *et al.*, "Model predictive control with signal temporal logic specifications," in *Proc. Int. Conf. Decision Control*, Dec. 2014, pp. 81–87.

[91] A. Girard, G. Pola, and P. Tabuada, "Approximately bisimilar symbolic models for incrementally stable switched systems," *IEEE Trans. Autom. Control*, vol. 55, no. 1, pp. 116–126, Jan. 2010.

[92] J. Mazo, M. A. Davitian, and P. Tabuada, "PESSOA: A tool for embedded controller synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, 2010, vol. 6174, pp. 566–569.

[93] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Model based synthesis of control software from system level formal specifications," *Trans. Softw. Eng. Methodol.*, vol. 23, no. 1, 2014, Art. ID. 6.

[94] I. Saha, R. Ramaithitima, V. Kumar, G. J. Pappas, and S. A. Seshia, "Automated composition of motion primitives for multi-robot systems from safe LTL specifications," in *Proc. IEEE/RSJ IROS*, Sep. 2014, pp. 1525–1532.

[95] D. Bresolin *et al.*, "Open problems in verification and refinement of autonomous robotic systems," in *Proc. Euromicro DSD*, Sep. 2012, pp. 469–476.

[96] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *Proc. Symp. Theor. Aspects Comput. Sci.*, 1995, pp. 229–242.

[97] G. Behrmann *et al.*, "UPPAAL-Tiga: Time for playing games!" in *Proc. Int. Conf. Comput.-Aided Verification*, 2007, pp. 121–125.

[98] UPPAAL-Tiga, a synthesis tool for timed games. [Online]. Available: http://people.cs.aau.dk/adavid/tiga/.

[99] C. Tomlin, J. Lygeros, and S. Sastry, "A game theoretic approach to controller design for hybrid systems," *Proc. IEEE*, vol. 88, no. 7, pp. 949–970, Jul. 2000.

[100] A. Balluchi, L. Benvenuti, T. Villa, H. Wong-Toi, and A. L. Sangiovanni-Vincentelli, "Controller synthesis for hybrid systems with a lower bound on event separation," *Int. J. Control*, vol. 76, no. 12, pp. 1171–1200, Aug. 2003.

[101] H. Wong-Toi, "The synthesis of controllers for linear hybrid automata," in *Proc. IEEE Conf. Decision Control*, Dec. 1997, vol. 5, pp. 4607–4612.

[102] M. Benerecetti, M. Faella, and S. Minopoli. "Automatic synthesis of switching controllers for linear hybrid systems: Safety control," *Theor. Comput. Sci.*, vol. 493, pp. 116–138, 2013.

[103] D. Bresolin, L. Di Guglielmo, L. Geretti, and T. Villa, "Correct-by-construction code generation from hybrid automata specification," in *Proc. IWCMC*, Jul. 2011, pp. 1660–1665.

[104] L. Di Guglielmo, S. Seshia, and T. Villa, "Synthesis of implementable control strategies for lazy linear hybrid automata," in *Proc. FedCSIS*, Sep. 2013, pp. 1381–1388.

[105] M. D. Wulf, "From timed models to timed implementations," Ph.D. dissertation, Universite Libre de Bruxelles, Brussels, Belgium, 2006–2007.

[106] M. De Wulf, L. Doyen, and J.-F. Raskin, "Almost ASAP semantics: From timed models to timed implementations," in *Hybrid Systems: Computation and Control*, vol. 2993, R. Alur and G. Pappas, Eds. Berlin, Germany: Springer-Verlag, 2004, pp. 296–310.

[107] M. Agrawal and P. Thiagarajan, "The discrete time behavior of lazy linear hybrid

automata," in *Hybrid Systems: Computation and Control*, vol. 3414, M. Morari and L. Thiele, Eds. Berlin, Germany: Springer-Verlag, 2005, pp. 55–69.

[108] P. Nuzzo, J. Finn, A. Iannopollo, and A. L. Sangiovanni-Vincentelli, "Contract-based design of control protocols for safety-critical cyber-physical systems," in *Proc. Design, Autom. Test Eur.*, Mar. 2014, pp. 1–4.

[109] I. Moir and A. Seabridge, *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration* 3rd ed., Chichester, England: Wiley, 2008.

[110] IBM ILOG CPLEX Optimizer, Feb. 2012. [Online]. Available: http://www.ibm.com/software/integration/optimization/cplex-optimizer/.

[111] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu, "Assume-guarantee verification for probabilistic systems," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, vol. 6015, J. Esparza and R. Majumdar, Eds. Berlin, Germany: Springer-Verlag, 2010, pp. 23–37.

[112] B. Caillaud *et al.*, "Compositional design methodology with Constraint Markov Chains," in *Proc. 7th QEST*, Sep. 2010, pp. 123–132.

[113] G. Gössler, D. N. Xu, and A. Girault. "Probabilistic contracts for component-based design," *Formal Methods Syst. Design*, vol. 41, no. 2, pp. 211–231, 2012.

[114] M. Kwiatkowska, G. Norman, and D. Parker, "Stochastic model checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, vol. 4486, M. Bernardo and J. Hillston, Eds. New York, NY, USA: Springer-Verlag, 2007, pp. 220–270.

[115] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. Int. Conf. Comput.-Aided Verification*, vol. 6806, G. Gopalakrishnan and S. Qadeer, Eds. New York, NY, USA: Springer-Verlag, 2011, pp. 585–591.

ABOUT THE AUTHORS

Pierluigi Nuzzo (Student Member, IEEE) received the Laurea (M.Sc.) degree in electrical engineering (*summa cum laude*) from the University of Pisa, Pisa, Italy, in 2003, and the Diploma in engineering (*summa cum laude*) from the Sant'Anna School of Advanced Studies, Pisa, in 2004, and is currently pursuing the Ph.D. degree in electrical engineering and computer sciences at the University of California, Berkeley, CA, USA.

Before joining the University of California at Berkeley, he was a Researcher with IMEC, Leuven, Belgium, working on the design of energy-efficient A/D converters and frequency synthesizers for reconfigurable radio. During the summer of 2002, he was with the Fermi National Accelerator Laboratory, Batavia, IL, USA, working on ASIC testing. From 2004 to 2006, he was with the Department of Information Engineering, University of Pisa, and with IMEC as a visiting scholar, working on low-power A/D converter design for wideband communications and design methodologies for mixed-signal integrated circuits. His research interests include: methodologies and tools for cyber-physical system and mixed-signal system design; contracts, interfaces, and compositional methods for embedded system design; energy-efficient analog and mixed-signal circuit design.

Mr. Nuzzo received First Place in the operational category and Best Overall Submission in the 2006 DAC/ISSCC Design Competition, a Marie Curie Fellowship from the European Union in 2006, the University of California at Berkeley EECS departmental fellowship in 2008, the University of California at Berkeley Outstanding Graduate Student Instructor Award in 2013, and the IBM Ph.D. Fellowship in 2012 and 2014.

Alberto L. Sangiovanni-Vincentelli (Fellow, IEEE) received the Laurea degree (*summa cum laude*) in electrical engineering and computer sciences from the Politecnico di Milano, Milan, Italy, in 1971.

He currently holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences with the University of California, Berkeley, CA, USA. He was a co-founder of Cadence and Synopsys, the two leading companies in the area of electronic design automation (EDA), and the founder and Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems (PARADES) research center in Rome, Italy. He has been a member of the Board of Directors of Cadence, KPIT-Cummins, Sonics, and Expert Systems. He had been a member of the ST Microelectronics Advisory Board for 10 years. He was a member of the HP Strategic Technology Advisory Board (2005–2007), a member of the Science and Technology Advisory Board of General Motors (2003–2013), and he is a member of the Technology Advisory Council of United Technologies Corporation. He is also member of the Scientific Council of the Italian National Science Foundation (CNR). Since 2010, he has been a member of the Executive Committee of the Italian Institute of Technology. Since July 2012, he has been named Chairperson of the Comitato Nazionale Garanti per la Ricerca. He is an author of over 880 papers, 17 books, and three patents in the area of design tools and methodologies, large-scale systems, embedded systems, hybrid systems, and innovation.

Dr. Sangiovanni-Vincentelli has been an IEEE Fellow since 1982, a Member of the National Academy of Engineering since 1998, and an ACM Fellow since 2014. In 1981, he received the Distinguished Teaching Award of the University of California. He received the worldwide 1995 Graduate Teaching Award of the IEEE for "inspirational teaching of graduate students." In 2002, he was the recipient of the Aristotle Award of the Semiconductor Research Corporation. He received numerous research awards including the Guillemin-Cauer Award (1982–1983), the Darlington Award (1987–1988) of the IEEE for the best paper bridging theory and applications, two awards for the best paper published in the IEEE Transactions on Circuits and Systems and *Computer-Aided Design*, five best paper awards, and one best presentation award at the Design Automation Conference. In 2001, he was given the Kaufman Award of the Electronic Design Automation Council for "pioneering contributions to EDA." In 2008, he was awarded the IEEE/RSE Wolfson James Clerk Maxwell Medal "for pioneering innovation and leadership in electronic design automation that have enabled the design of modern electronics systems and their industrial implementation." In 2009, he was awarded an honorary Doctorate by the University of Aalborg, Aalborg, Denmark, and he received the first ACM/IEEE A. Richard Newton Technical Impact Award in Electronic Design Automation to honor persons for an outstanding technical contribution within the scope of electronic design automation. In 2012, he was awarded an honorary Doctorate from the Royal Institute of Technology (KTH), Stockholm, Sweden, and he received the Lifetime Achievement Award from EDAA.

Davide Bresolin received the Ph.D. degree in computer science from the University of Udine, Udine, Italy, in 2007.

He is an Assistant Professor with the Computer Science and Engineering Department, University of Bologna, Bologna, Italy. From 2007 to 2013, he was a Research Fellow with the Computer Science Department, University of Verona, Verona, Italy, where he collaborated with the Electronic Systems Design Group (ESD) and the ALTAIR Robotics Group. His research activity is focused on formal verification of cyber-physical and embedded systems using hybrid automata and temporal logics, on automata theory, and on temporal representation and reasoning using interval-based temporal logics. He is actively collaborating to the development of ARIADNE, an open-source tool for the numerical analysis of continuous and hybrid systems.

Luca Geretti was born in Udine, Italy, in 1979. He received the Laurea degree in electrical engineering and the Ph.D. degree in computer engineering from the University of Udine, Udine, Italy, in 2005 and 2009, respectively.

He was a Research Fellow with the Department of Computer Science, University of Verona, Verona, Italy, between 2009 and 2011. He is currently a Research Fellow with the Department of Electrical, Management and Mechanical Engineering, University of Udine. His current research interests are in the fields of formal verification and parallel and distributed computing.

Tiziano Villa studied mathematics at the University of Milano, Milan, Italy; University of Pisa, Pisa, Italy; and Cambridge University, Cambridge, U.K. He received the Ph.D. degree in electrical engineering and computer science (EECS) from the University of California, Berkeley, CA, USA, 1995.

From 1980 to 1985, he worked as a computer-aided design specialist with the integrated circuits division of the CSELT Labs, Turin, Italy. From 1986 to 1996, he was a Research Assistant with the Electronics Research Laboratory, University of California, Berkeley. From 1997 to 2001, he was a Research Scientist with the PARADES Labs, Rome, Italy. From 2002 to 2006, he was an Associate Professor with the Department of Electrical, Industrial and Mechanical Engineering (DIEGM), Universita' degli Studi di Udine, Udine, Italy. Since 2006, he has been a Full Professor with the Department of Computer Science, Università degli Studi di Verona, Verona, Italy. He coauthored the books *Synthesis of FSMs: Functional Optimization* (Kluwer, 1997), *Synthesis of FSMs: Logic Optimization* (Kluwer, 1997), and *The Unknown Component Problem: Theory and Applications* (Springer, 2012). He co-edited the book *Coordination Control of Distributed Systems* (Springer, 2015). His research interests include computer-aided design of digital circuits (especially logic synthesis), formal verification, cyber-physical systems, and automata theory.

In 1991, he was awarded the Tong Leong Lim Pre-doctoral Prize at the EECS Department of the University of California, Berkeley.