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

Pierluigi Nuzzo, Alberto Sangiovanni-Vincentelli, Davide Bresolin, Luca Geretti, 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 abstractions. 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.**

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 health-care 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* (CPS) [1], [2], [3] networks 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 CPS 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 sub-disciplines, 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 requirement capture and designers' experience can lead to implementations that are inefficient and sometimes do not even satisfy the requirements, yielding long re-design 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], 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 CPS 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 CPS. However, an all-encompassing framework for CPS design that helps interconnect 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 complex-

P. Nuzzo and A. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, USA. Email: {nuzzo, alberto}@eecs.berkeley.edu.

D. Bresolin is with the Department of Computer Science and Engineering, University of Bologna, Bologna, Italy. Email: davide.bresolin@unibo.it.

L. Geretti is with the Department of Electrical Engineering (DIEGM), University of Udine, Udine, Italy. Email: luca.geretti@univr.it.

T. Villa is with the Department of Computer Science, University of Verona, Verona, Italy. Email: tiziano.villa@univr.it.

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 through the EU project FP7-ICT-223844 CON4COORD.

[1] see e.g., http://www.autorecalls.us

[2] see, e.g., http://en.wikipedia.org/wiki/Boeing_787

ity 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 pre-designed 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 (e.g. captured using linear temporal logic), to the one of hybrid systems, (e.g. captured using signal temporal logic or networks of hybrid automata). 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 relation 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. 1 (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 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 extensively contracts 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 *top-down phase* of each design step, 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 into 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. In the *bottom-up phase*, we build and model the component library (including both plant and controller).

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: (i) a set of requirements are *consistent*, i.e. there exists an implementation satisfying all of them; (ii) an aggregation of components are *compatible*, i.e. there exists an environment in which they can correctly operate; (iii) 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 CPS. 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 is 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 CPS, 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], [10], analog integrated systems [5] and, more recently, synthesis and verification of control algorithms for CPS [13], [14].

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], [15]. 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 we 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 $u \in \mathcal{U}$, output $y \in \mathcal{Y}$ and internal $x \in \mathcal{X}$ *variables* (including state variables); a set of configuration *parameters* $\kappa \in \mathcal{K}$, and a set of input, output and bidirectional *ports* $\lambda \in \mathcal{L}$. 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 and internal variables given the values of the input variables and configuration parameters. 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, to simplify, we also use M to denote the set of behaviors of a component;
- a set of *non-functional models*, i.e. maps that allow computing non-functional attributes of a component, corresponding to particular valuations of its input variables and configuration parameters. Examples of non-functional 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 $l + 1$. 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 κ are fixed. Then, a *contract* $\mathcal{C}$ for a component M is a triple (V, A, G), where $V = \{u, y, x\}$ 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 $E \subseteq A$.

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 $G' = G \cup \overline{A}$. Therefore, 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:

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \quad (1)$$

$$G = G_1 \cap G_2. \quad (2)$$

The composite contract must satisfy the guarantees of both, which explains the operation of intersection in (2) [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 (1) and (2), we obtain $G = (y > 0)$ and $A = (y > 0) \vee (y \leq 0) = \mathtt{T}$, 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 (1), 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:

$$\begin{aligned} A &= \max\{A' | A' \subseteq A_1 \cup \overline{G_2}, A' \subseteq A_2 \cup \overline{G_1}\} \\ &= (A_1 \cup \overline{G_2}) \cap (A_2 \cup \overline{G_1}) \\ &= (A_1 \cap A_2) \cup (A_1 \cap \overline{G_1}) \cup (A_2 \cap \overline{G_2}) \cup (\overline{G_1} \cap \overline{G_2}) \\ &= (A_1 \cap A_2) \cup \overline{G_1} \cup \overline{G_2}, \end{aligned} \quad (3)$$

which reduces to (1). The last equality in (3) 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 $V = V_1 \cup V_2$.

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. 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 $G \neq \emptyset$. 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).

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 $A \supseteq A'$ and $G \subseteq G'$. 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 $\mathcal{C}$.

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.

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. (i) $\mathcal{C}_1 \wedge \mathcal{C}_2$ is guaranteed to refine both $\mathcal{C}_1$ and $\mathcal{C}_2$, and (ii) 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

$$\mathcal{C}_1 \wedge \mathcal{C}_2 = (A_1 \cup A_2, G_1 \cap G_2). \tag{4}$$

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 [16]. Instead, composition is replaced with the concept of *circular reasoning* [17]: 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*; 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 exemplify this concept, consider the problem of mapping a specification platform of a system at level $l+1$ onto 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 does not directly match the implementation platform architecture. In particular, let $\mathcal{C} = \bigotimes_{i \in I} \left(\bigwedge_{k \in K_i} \mathcal{C}_{ik} \right)$ and $\mathcal{M} = \bigotimes_{j \in J} \left(\bigwedge_{l \in L_j} \mathcal{M}_{jl} \right)$ be two contracts respectively describing the specification and implementation platforms. In this example, we have defined $\mathcal{C}$ and $\mathcal{M}$ out of a composition of I and J components, respectively. The contract for each component is then defined out of a conjunction of simpler contracts. Since the components of $\mathcal{M}$ and $\mathcal{C}$ may not directly match, checking that $\mathcal{M} \preceq \mathcal{C}$ in a compositional way, by reasoning on the components of $\mathcal{M}$ and $\mathcal{C}$ independently, may not be effective.

However, it is still possible to model the mapping of the specification over the implementation by the composition $\mathcal{C} \otimes \mathcal{M}$. This composition captures the fact that the actual satisfaction of all the design requirements and viewpoints by a deployment depends on the supporting execution platform and on the way system functionalities are mapped to it. In the composition, assumptions made from the specification platform on the implementation platform get discharged by the guarantees of the implementation platform, and *vice versa*, as indicated by (1) and (2). Refinement can then be checked by checking instead that $\mathcal{C} \otimes \mathcal{M} \preceq \mathcal{C}$, which can be performed compositionally, by directly matching the components of $\mathcal{C}$ with the ones of $\mathcal{C} \otimes \mathcal{M}$. The composite contract $\mathcal{C} \otimes \mathcal{M}$ is a *vertical contract*, used to formalize mechanisms for mapping a specification over an execution platform, such as the ones adopted in the METROPOLIS [18], METROII [19], and, more recently, the METRONOMY frameworks [20].

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 [14]. 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, 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

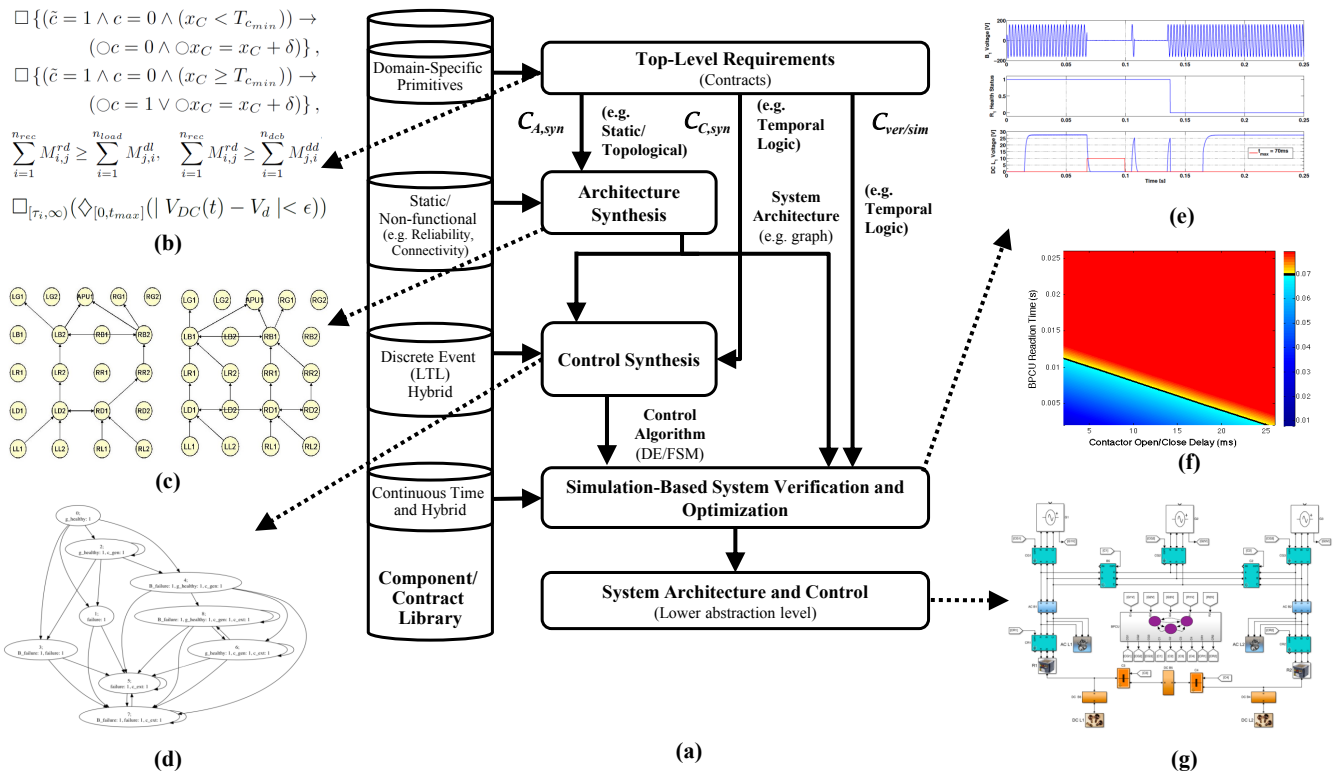

Fig. 1. (a) Structure of the proposed contract-based methodology for CPS design, from top-level requirements to the definition of system architecture and control algorithm. Demonstration of the different design steps on the aircraft electric power system example in Section VI [13]: (b) requirement formalization; (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.

system, as well as the different levels of abstraction in the design flow. As shown in Fig. 1 (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 within the European projects SPEEDS and CESAR [12], linked to automata-based formalisms, or to the higher-level *domain-specific language* (DSL) proposed in [13], and further exemplified in Section VI.

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, 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-Theories (SMT) solvers [21], [22] or mathematical optimization software, such as mixed integer-linear, mixed integer-semidefinite-positive, or mixed integer/non-linear 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. In particular, linear temporal logic (LTL) [23] 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. 1a).

Signal temporal logic (STL) [24] can deal with dense-time real signals and continuous dynamical models (continuous abstraction in Fig. 1a). 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) [25] and *hybrid automata* [26]. 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 '80s 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 "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 CPS. 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 $\sigma = s_0 s_1 s_2 \dots$ 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 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 consists in adding operators to express timing constraints between discrete events. This leads to the development of *real-time temporal logics* such as Metric Temporal Logic (MTL) [27]. For instance, real-time temporal logics can express properties like "An alert must be resolved in 10 seconds", 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 [24], which is able to express properties like "If the temperature reaches 90 degrees then it must eventually decrease below 60". Such a property can be formalized by the formula $\Box(t \geq 90 \rightarrow \Diamond\, t < 60)$, which constrains any time instant where the temperature t is greater or equal to 90 to be followed by a time instant where the temperature is below 60.

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 [25], which extends the LTL with regular expressions (RE), and Differential Dynamic Logic (d$\mathcal{L}$) [28], 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 $\Box(t \geq 90 \rightarrow \bigcirc\, alert)$, where $\bigcirc$ is the "next discrete event" operator. On the other hand, the hybrid property "for the state of a train controller *train*, the property $z \leq 100$ always holds true when starting in a state where $v^2 \leq 10$ is true", can be expressed by the 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 [15]. 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 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* [29];
- *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 [25].

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 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. 2 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 $\frac{T}{2}$. The generator will then switch

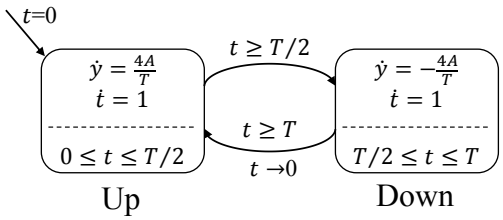

Fig. 2. Hybrid automaton specifying a triangle wave generator.

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 non-trivial 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 [26]. The class of *hybrid input/output automata* enables compositional analysis of systems [30]. In *timed automata* [31] 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* [32] extend timed automata by allowing piecewise constant dynamics, while still keeping decidability of the reachability problem. *Linear hybrid automata* [33] 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 in [34]. 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 non-deterministic 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. Based on the results in [34], for uniform assumptions and safety guarantees, 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 over-approximations and under-approximations of the evolution, a capability supported by very few tools.

C. Verification Tools

As shown in Section III-A and Section III-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 [35], which verifies real-time systems modeled by timed automata with respect to requirements expressed in the real-time logic TCTL (Timed Computation Tree Logic), using a backward-forward analysis approach.

The same approach was then extended to support rectangular automata in HYTECH [36], 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 [37], 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 [38].

2) Tools Based on Reachable Set Approximations: When the dynamics is 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, *over-approximations* may be used to obtain positive answers, while *under-approximations* give negative answers.

One of the first tools that enabled verification of hybrid systems with complex dynamics is *d/dt* [39]. 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 [40] 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 [41] 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, non-deterministic 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* [42] supports systems with non-linear 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 [43], [34] 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 non-linear 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 [44]. PYHYBRIDANALYSIS [45] 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 [46], [47]. 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 in [47] has been implemented by CHECKMATE [48], 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 non-linear systems is HSOLVER [49], 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 non-linear 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 [50] 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 [51] 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 over-approximation 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 logics 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 robust tool using theorem proving techniques in the context of hybrid systems is KEYMAERA [28], 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 (in fact, 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 [52], a MATLAB/C++ toolbox for the simulation, verification of temporal logic properties and reachability analysis of dynamical systems, defined as systems of ordinary differential equations (ODEs) or by external modeling tools such as SIMULINK. It uses systematic simulation to compute an under-approximation 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 [53] 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) [54] 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. 1 (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 [55]. 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*. 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 multi-view 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 [14]. 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[3] and PTOLEMY II[4];
- Hardware description languages, such as Verilog[5], VHDL[6], or transaction-level modeling languages, such as SystemC[7], together with their respective analog-mixed-signal extensions[8];
- Modeling languages specifically tailored for acausal multi-physics systems, such as Modelica[9], supported by tools such as DYMOLA[10] or JMODELICA[11];
- Languages for architecture modeling, such as the Systems Modeling Language (SysML)[12] and the Architecture Analysis & Design Language (AADL)[13].

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 CPS. One of the first examples of these languages is Charon [56]. 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 [57] 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) [58]. 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) [59]. 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 [60], which also accounts for implementation considerations, such as equation sorting and event detection, and the interchange format for switched linear systems defined in [61]. More recently, the Compositional Interchange Format (CIF) has been proposed to overcome some of the limitations of previous languages [62], such as the absence of hierarchy in HSIF, and the limitation to linear dynamics only in [61]. 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 re-use 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. [63] propose the customization of SysML [64] 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 [65], [66]. Initially developed within the MODELISAR project, and currently supported by a number of industrial partners and tools[14], FMI shows promise for enabling the exchange and interoperation of model components. The FMI standard supports both co-simulation, where a component called FMU (Functional Mock-up Unit) implements its own simulation algorithm, 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 non-deterministic and unexpected behaviors, are currently object of investigation [67].

V. MAPPING SPECIFICATIONS TO IMPLEMENTATIONS

In the absence of a unified framework for automated synthesis of CPS 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

[3]http://www.mathworks.com/products/simulink

[4]http://ptolemy.eecs.berkeley.edu

[5]http://www.verilog.com/

[6]http://www.vhdl.org

[7]http://www.accellera.org/downloads/standards/systemc

[8]http://www.eda.org/verilog-ams/, http://www.eda.org/vhdl-ams/, http://www.accellera.org/downloads/standards/systemc/ams

[9]https://www.modelica.org/

[10]www.dynasim.se/

[11]http://www.jmodelica.org/

[12]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).

[13]http://www.aadl.info/aadl/currentsite

[14]https://www.fmi-standard.org/

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 $\mathcal{C}_{syn}$ a contract that can be used as input of a specialized synthesis tool, and as $\mathcal{C}_{opt}$ a contract that serves as a conjunction of constraints in a more generic optimization problem. $\mathcal{C}_{opt}$ can be further characterized as $\mathcal{C}_{ver} \wedge \mathcal{C}_{sim}$, where $\mathcal{C}_{ver}$ denotes a contract whose satisfaction can be formally verified, e.g. using the tools introduced in Section III, while $\mathcal{C}_{sim}$ 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, $\mathcal{C}_{A,syn}$ in Fig. 1 (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 [13]. As shown in [13], the formulation above encompasses a variety of requirements, such as connectivity, safety, reliability, and energy balance. These requirements are mapped on a representation of the system architecture, e.g. in terms of a labelled graph, where nodes represent the (parameterized) components and edges represent their interconnections.

To handle reliability requirements, the ARCHEX framework [13], [68] 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, following an approach similar to [69], [22]. The solver iteratively provides candidate configurations that are analyzed and accordingly modified, only when needed, 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 eagerly generates a monolithic problem instance in polynomial time, using 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. $\mathcal{C}_{C,syn}$ in Fig. 1 (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 CPS.

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 [70], [71], [72], [73]. 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 $\mathcal{C}_{LTL}$ an LTL contract of the form $(\varphi_e, \varphi_e \rightarrow \varphi_s)$, 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 $\mathcal{C}_{LTL}$ and a controlled plant that tries to satisfy it. A control strategy is a partial function $f : (s_0 s_1 \dots s_{t-1}, e_t) \mapsto d_t$, 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 [70]. 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 [74], [73], [75], [76], [77].

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 [78], [79], as available for instance in the third revision of the CIF language for supervisory control synthesis [80]. More recently, a synthesis method for discrete-time CPS subject to STL specifications has been proposed based on a model predictive control framework [81], [82]. The 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 [83] has been recently introduced to obtain correct and complete abstractions of differential equations that can be used to solve controller design problems. PESSOA [84] 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 [85]. 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 multi-robot systems [86]. 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 [87]. Even in the context of timed automata, where the synthesis problem is known to be solvable in an exact way [88], efficient and practical tools are lacking. One of the few exceptions is UPPAAL-TIGA [89], [90], 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 (see [91], [92] for a general formulation). Some examples are the symbolic semi-algorithm to compute the controllable region of a linear hybrid automaton with respect to a safety goal described in [93], and the procedure to synthesize the maximal safe controller for more general hybrid systems with a lower bound on event separation reported in [92]. One of the few publicly available tools implementing this two-person game approach is PHAVER+ [94], an extension of PHAVER that can automatically synthesize discrete controllers for linear hybrid automata with respect to safety and reachability goals.

In [95] and [96] two synthesis (mapping) approaches are presented that can incorporate finite-precision sensors and actuators as well as the finite response time of the controller. The synthesis problem is addressed for two sub-classes 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. They were introduced in [97], [98], together with a parametric semantics for timed controllers called the Almost ASAP semantics, which relaxes the standard idealized ASAP (As Soon As Possible) 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. A corresponding automated tool chain, reported in [95], can extract from an elastic controller a correct-by-construction HW/SW implementation described in SystemC. On the other hand, lazy linear hybrid automata [99] 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 is discussed in [96].

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 $\mathcal{C}_{sim} = (\phi_e, \phi_e \rightarrow \phi_s)$ 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 *multi-objective 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,

$$\min_{\kappa \in \mathcal{K}, \pi \in \Pi} \quad C(\kappa, \pi) \tag{5}$$
$$\text{s.t.} \begin{cases} \mathcal{F}(s, \kappa) = 0 \\ \\ s \models \phi_s(\pi) \qquad \forall s \text{ s.t. } s \models \phi_e(\pi) \end{cases}$$

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' is shorthand notation for $s'(t) = (u'(t), y'(t), x'(t))$, the trace of input, output and internal signals (here represented as vectors of traces over time $t \in \mathbb{R}_+$) that are obtained by simulating the behavioral model $\mathcal{F}(.)$, defined in Section II-C. A multi-objective 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 $\mathcal{C}_{ver} = (\phi_e, \phi_e \rightarrow \phi_s)$ whose satisfaction can still be efficiently verified via formal methods, even if the synthesis problem is intractable. Moreover, it can be used

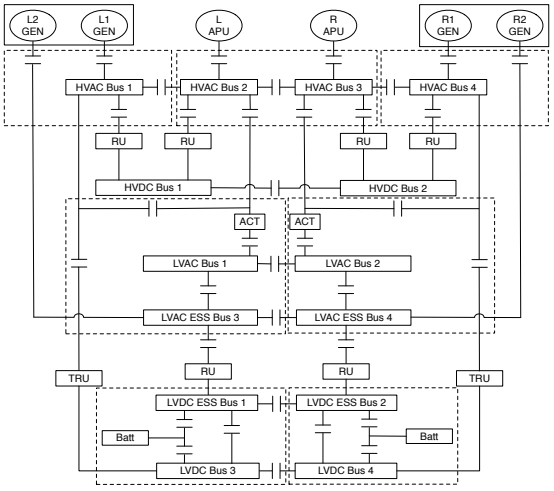

Fig. 3. Single-line diagram of an aircraft electric power system (Figure from [13]).

to perform joint design exploration of the controller and its execution platform, while guaranteeing that their specifications, captured by vertical contracts, are consistent. Typically, the controller requirements are defined in terms of several aspects that are related to the execution platform, including the timing behavior of the control tasks and of the communication between tasks, their jitter, the accuracy and resolution of the computation, and, more generally, requirements on power and resource consumption. These requirements are taken as assumptions by the controller, which in turn provides guarantees in terms of the amount of requested computation, activation times and data dependencies. As mentioned in Section II-B, 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 [13], [100].

Figure 3 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 [101]. Generators (e.g., two on the left and two on the right side of the aircraft, denoted as GEN in Fig. 3) deliver power to the loads (e.g. avionics, lighting, heating and motors, not represented in Fig. 3) via high-voltage and low-voltage AC (HVAC, LVAC) and DC buses (HVDC, LVDC), while the Auxiliary Power Units (APU) or batteries (Batt) are used when one of the generators fails. Essential buses (ESS in Fig. 3) supply loads that cannot be unpowered for more than a predefined period t_{max}, while non-essential 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. 3) 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. 1. 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 summary is based on the results in [13].

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 domain-specific language (DSL), which enables automatic translation of the specifications from a set of pre-defined primitives to one of the back-end formalisms mentioned in Section III. The proposed DSL can smoothly interface with pre-existing tools, such as visual programs for single-line diagrams, typically used by system engineers. Representative examples of system assumptions and guarantees are provided below.

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. The probability r_S is the reliability level of the system. To allow formalizing this requirement, a set of environment assumptions 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 any component fails during the flight, it will not come back online.

G_1. Reliability Level: 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) must be smaller than the system reliability level r_S.

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. 1 (b). 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 $\mathcal{C}_T \otimes \mathcal{C}_C \preceq \mathcal{C}_S$. While this proof is performed manually in [13], reasoning with contracts is still instrumental to co-designing architecture and control. In particular, Propositions 6.1 and 6.2 of [13] 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 below, can be *independently* deployed, while guaranteeing that the assembled system is correct and satisfies $\mathcal{C}_S$.

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

The *plant architecture* is modelled as a graph, where each node represents a component (with the exception of contactors, which are associated with edges) and each edge represents an interconnection. At this level of abstraction, the platform library $\mathcal{L}$ includes, as attributes, generator power ratings, component costs and failure probabilities, in addition to interconnection rules. The *supervisory controller* consists of one or more finite state machines, and is characterized by the reaction time T_r.

The safety, connectivity, power flow, and reliability requirements in $\mathcal{C}_T$ (both assumptions and guarantees) can be expressed as 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 trade-off between redundancy and cost can then be explored using ARCHEX, and the synthesized topology is offered as a specification for the control design step. As shown in Fig. 1 (c), the ILP-MR algorithm implemented in ARCHEX using CPLEX [102] as a back-end optimization engine is able to generate, in a few seconds, architectures for the primary distribution of an electric power system (including two generators, two AC buses, two rectifiers, two DC buses and two loads on each side) for different reliability requirements.

C. Control Design

Controller requirements can be defined as a contract $\mathcal{C}_C = (A_C, G_C)$, where A_C encodes the allowable behaviors of the environment (including the physical plant) and G_C encodes the desired behaviors of the closed-loop system, i.e. the top-level requirements. $\mathcal{C}_S$ can then be expressed as the heterogeneous conjunction between an LTL contract $\mathcal{C}_{LTL}$ and an STL contract $\mathcal{C}_{STL}$. The STL formulas in $\mathcal{C}_{STL}$ can either be obtained by heterogeneous refinement of a subset of LTL formulas in $\mathcal{C}_{LTL}$ 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.

As shown in Fig. 1 (a), $\mathcal{C}_{LTL}$ 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, as shown in Fig. 1 (d), using reactive synthesis techniques. The resulting controller will satisfy $\mathcal{C}_{LTL}$ by construction. Satisfaction of $\mathcal{C}_{STL}$ is then assessed on a hybrid model, including both the controller and an equation-based representation of the plant, by monitoring simulation traces while optimizing a set of system parameters. The resulting optimal controller configuration is returned as the final design, as represented in Fig. 1 (g).

Since the formulas in $\mathcal{C}_{LTL}$ are within the GR(1) fragment of LTL, a control protocol can be automatically synthesized using the TULIP Toolbox [73]. For the topologies explored in Section VI-B, a set of centralized and distributed control protocols were synthesized for a reliability level $r_S = r_T$ in approximately 0.5 to 2 s, for a number of states ranging from 4 to 113. A hybrid model implemented in SIMULINK, based on blocks from the SimPowerSystems library, as shown in Fig. 1 (g), is instead used to analyze and optimize the real-time performance of the controller, imported as a MATLAB function. The plant model includes the effects of non-ideal contactor response, implementing a fixed delay T_d to the open/close commands from the controller. It is then possible to explore the T_r-versus-T_d design space and find the maximum allowed controller 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 (5), where the constraints are expressed as a conjunction of parameterized STL formulas (with parameter $\pi = T_r$). In this case, the system behavior is the trace $s = (u, V_{DC})$, 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 in $\mathcal{C}_C$. The BREACH toolbox [52] was used to post-process the simulation traces and verify the satisfaction of STL formulas.

As an example, for the architecture in Fig. 1 (g), Fig. 1 (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 worst case scenario of cascaded faults in generators G_1, G_2 and rectifier R_1 [100]. 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. 1 (f) by sampling the parameter space in approximately 4 hours to populate a 13×13 point grid. 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 with the hard threshold $t_{max} = 70$ ms, thus providing the designer with a "safe" region (marked in blue in Fig. 1 (f)) for selecting the controller clock 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 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, modelled as networks of hybrid automata. To illustrate the application of the methodology, we used a concrete example from controller design in aircraft electric power systems.

Inspired by the design examples, 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.

Finally, we observe that several parameters impacting the behavior of CPS are subject to variability due to manufacturing tolerances, usage and faults. Moreover, the models that are normally used to design multi-physics systems inevitably introduce inaccuracies [103]. 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 [104], [105], [106]) is deemed as essential to improve on the scalability of stochastic analysis and synthesis techniques (e.g. see [107], [108]), and make their adoption actually feasible in current design flows.

References

[1] J. Sztipanovits, "Composition of cyber-physical systems," in *Proc. IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems*, March 2007, pp. 3–6.

[2] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. IEEE Int. Symposium on Object Oriented Real-Time Distributed Computing*, 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*, ser. Lecture Notes in Computer Science, S. Bensalem, Y. Lakhneck, and A. Legay, Eds. Springer Berlin Heidelberg, 2014, vol. 8415, pp. 193–208. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54848-2_13

[4] A. Sangiovanni-Vincentelli, "Quo vadis, SLD? Reasoning about the trends and challenges of system level design," *Proc. IEEE*, no. 3, pp. 467–506, 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: The MIT Press, 2008.

[7] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects*. Springer-Verlag, 2008, pp. 200–225.

[8] L. de Alfaro and T. A. Henzinger, "Interface automata," in *Proc. Symp. Foundations of Software Engineering*. ACM Press, 2001, pp. 109–120.

[9] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18-3, no. 3, pp. 217–238, 2012.

[10] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier *et al.*, "Contracts for System Design," INRIA, Rapport de recherche RR-8147, Nov. 2012.

[11] M. Masin, A. Sangiovanni-Vincentelli, A. Ferrari, L. Mangeruca, H. Broodney, L. Greenberg, M. Sambur, D. Dotan, S. Zolotnizky, and S. Zadorozhniy, "META II: Lingua franca design and integration language," 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, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *Proc. Design, Automation and Test in Europe*, Mar. 2011, pp. 1–6.

[13] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donzé, and S. Seshia, "A contract-based methodology for aircraft electric power system design," *IEEE Access*, vol. 2, pp. 1–25, 2014.

[14] A. Iannopollo, P. Nuzzo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Library-based scalable refinement checking for contract-based design," in *Proc. Design, Automation and Test in Europe*, Mar. 2014.

[15] P. Nuzzo, A. Iannopollo, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Are interface theories equivalent to contract theories?" in *Int. Conf. Formal Methods and Models for Co-Design*, Oct. 2014.

[16] S. Graf, R. Passerone, and S. Quinton, "Contract-based reasoning for component systems with rich interactions," in *Embedded Systems Development*, ser. Embedded Systems, A. Sangiovanni-Vincentelli, H. Zeng, M. Di Natale, and P. Marwedel, Eds. Springer New York, 2014, vol. 20, pp. 139–154. [Online]. Available: http://dx.doi.org/10.1007/978-1-4614-3879-3_8

[17] R. Alur and T. Henzinger, "Reactive modules," *Formal Methods in System Design*, vol. 15, no. 1, pp. 7–48, 1999. [Online]. Available: http://dx.doi.org/10.1023/A%3A1008739929481

[18] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, and Y. Watanabe, "Metropolis: an integrated electronic system design environment," *Computer*, vol. 36, no. 4, 2003.

[19] F. Balarin, A. Davare, M. D'Angelo, D. Densmore, T. Meyerowitz, R. Passerone, A. Pinto, A. Sangiovanni-Vincentelli, A. Simalatsar, Y. Watanabe, G. Yang, and Q. Zhu, "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, London, New York: CRC Press, Taylor and Francis Group, November 2009, ch. 10, p. 259.

[20] L. Guo, Z. Qi, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee, "Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems," in *Proc. Int. Conf. Hardware-Software Codesign and System Synthesis*, Oct. 2014.

[21] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Satisfiability Modulo Theories, Chapter in Handbook of Satisfiability*. IOS Press, 2009.

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

[23] A. Pnueli, "The temporal logic of programs," in *Annual Symp. on Foundations of Computer Science*, Nov. 1977, pp. 46–57.

[24] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Modeling and Analysis of Timed Systems*, 2004, pp. 152–166.

[25] A. Cimatti, M. Roveri, and S. Tonetta, "Requirements validation for hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds. Springer Berlin Heidelberg, 2009, vol. 5643, pp. 188–203.

[26] 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 Systems*, ser. LNCS, vol. 736. Springer, 1993, pp. 209–229.

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

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

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

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

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

[32] 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. [Online]. Available: http://dx.doi.org/10.1006/jcss.1998.1581

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

[34] L. Benvenuti, D. Bresolin, P. Collins, A. Ferrari, L. Geretti, and T. Villa, "Assume-guarantee verification of nonlinear hybrid systems with ARIADNE," *Int. J. Robust Nonlinear Control*, vol. 24, no. 4, pp. 699–724, 2014.

[35] S. Yovine, "KRONOS: a verification tool for real-time systems," *Int. J. Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 123–133, 1997. [Online]. Available: http://dx.doi.org/10.1007/s100090050009

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

[37] 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. [Online]. Available: http://dx.doi.org/10.1002/spe.1006

[38] 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.

[39] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Hybrid Systems: Computation and Control*, ser. LNCS. Springer Berlin Heidelberg, 2000, vol. 1790, pp. 20–31. [Online]. Available: http://dx.doi.org/10.1007/3-540-46430-1_6

[40] G. Frehse, "PHAVer: algorithmic verification of hybrid systems past HyTech," *Int. J. Software Tools for Technology Transfer*, vol. 10, pp. 263–279, 2008.

[41] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. LNCS. Springer Berlin / Heidelberg, 2011, vol. 6806, pp. 379–395.

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

[43] L. Benvenuti, D. Bresolin, P. Collins, A. Ferrari, L. Geretti, and T. Villa, "Ariadne: Dominance checking of nonlinear hybrid automata using reachability analysis," in *Reachability Problems*, ser. Lecture Notes in Computer Science, A. Finkel, J. Leroux, and I. Potapov, Eds. Springer Berlin Heidelberg, 2012, vol. 7550, pp. 79–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33512-9_8

[44] A. Casagrande, C. Piazza, and A. Policriti, "Discrete semantics for hybrid automata," *Discrete Event Dynamic Systems*, vol. 19, no. 4, pp. 471–493, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10626-009-0082-7

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

[46] R. Alur, T. Dang, and F. Ivančić, "Counterexample-guided predicate abstraction of hybrid systems," *Theoretical Computer Science*, vol. 354, no. 2, pp. 250 – 271, 2006.

[47] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald, "Abstraction and counterexample-guided refinement in model checking of hybrid systems," *Int. J. Found. Comput. Sci.*, vol. 14, no. 4, pp. 583–604, 2003. [Online]. Available: http://dx.doi.org/10.1142/S012905410300190X

[48] B. I. Silva, K. Richeson, B. H. Krogh, and A. Chutinan, "Modeling and verification of hybrid dynamical system using CheckMate," in *ADPM*, 2000.

[49] S. Ratschan and Z. She, "Safety verification of hybrid systems by constraint propagation based abstraction refinement," *ACM Transactions in Embedded Computing Systems*, vol. 6, no. 1, 2007.

[50] A. Tiwari, "Abstractions for hybrid systems," *Formal Methods in System Design*, vol. 32, no. 1, pp. 57–83, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10703-007-0044-3

[51] A. Cimatti, S. Mover, and S. Tonetta, "SMT-based scenario verification for hybrid systems," *Formal Methods in System Design*, vol. 42, no. 1, pp. 46–66, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10703-012-0158-0

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

[53] 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 and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 254–257.

[54] T. Mancini, F. Mari, A. Massini, I. Melatti, F. Merli, and E. Tronci, "System level formal verification via model checking driven simulation," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, vol. 8044. Springer - Verlag, 2013, pp. 296–312.

[55] J. C. Willems, "The behavioral approach to open and interconnected systems," *Control Systems Magazine*, pp. 46–99, 2007.

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

[57] 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.

[58] 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 Software*. New York, NY, USA: ACM, 2006, pp. 132–141. [Online]. Available: http://doi.acm.org/10.1145/1176887.1176907

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

[60] A. Pinto, L. P. Carloni, R. Passerone, and A. L. Sangiovanni-Vincentelli, "Interchange format for hybrid systems: Abstract semantics," in *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006, Proceedings*. Springer, 2006, pp. 491–506.

[61] S. D. 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.

[62] 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.

[63] 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 Management*, 2009.

[64] *OMG Systems Modeling Language*. [Online]. Available: http://www.sysml.org/

[65] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," 2012.

[66] MODELISAR Consortium and Modelica Association, *Functional Mock-up Interface for Co-Simulation. Version 1.0*. Retrieved from https://www.fmi- standard.org, Oct. 2010.

[67] D. Broman, C. Brooks, L. Greenberg, E. A. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of FMUs for co-simulation," in *Proc. ACM IEEE Int. Conf. Embedded Software*.

Piscataway, NJ, USA: IEEE Press, 2013, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555754.2555756

[68] 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, Automation and Test in Europe*, Mar. 2015.

[69] 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.

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

[71] 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.

[72] 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.

[73] 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 Systems: Computation and Control*. New York, NY, USA: ACM, 2011, pp. 313–314.

[74] A. Pnueli, Y. Saar, and L. D. Zuck, "Jtlv: A framework for developing verification algorithms," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin Heidelberg, 2010, vol. 6174, pp. 171–174.

[75] B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem, "Anzu: A tool for property synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 258–262. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73368-3_29

[76] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Knighofer, M. Roveri, V. Schuppan, and R. Seeber, "RATSY a new requirements analysis tool with synthesis," in *Proc. Int. Conf. Comput.-Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin Heidelberg, 2010, vol. 6174, pp. 425–429. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_37

[77] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, G. Hofferek, B. Jobstmann, B. Knighofer, and R. Knighofer, "Synthesizing robust systems," *Acta Informatica*, vol. 51, no. 3-4, pp. 193–220, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00236-013-0191-5

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

[79] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, ser. SpringerLink Engineering. Springer, 2008.

[80] D. A. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. M. van de Mortel-Fronczak, and M. A. Reniers, "CIF 3: Model-based engineering of supervisory controllers," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Springer, 2014, pp. 575–580. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54862-8_48

[81] M. Maasoumy, P. Nuzzo, F. Iandola, M. Kamgarpour, A. Sangiovanni-Vincentelli, and C. Tomlin, "Optimal load management system for aircraft electric power distribution," in *Int. Conf. Decision and Control*, Dec 2013, pp. 2939–2945.

[82] V. Raman, A. Donze, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications," in *Int. Conf. Decision and Control*, Dec 2014.

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

[84] J. Mazo, Manuel, A. Davitian, and P. Tabuada, "PESSOA: A tool for embedded controller synthesis," in *Computer Aided Verification*, ser. LNCS, vol. 6174, 2010, pp. 566–569. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_49

[85] F. Mari, I. Melatti, I. Salvo, and E. Tronci, "Model based synthesis of control software from system level formal specifications," *ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY*, vol. 23, no. 1, p. Article 6, 2014.

[86] 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 Int. Conf. Intelligent Robots and Systems (IROS)*, Sep. 2014.

[87] D. Bresolin, L. Di Guglielmo, L. Geretti, R. Muradore, P. Fiorini, and T. Villa, "Open problems in verification and refinement of autonomous robotic systems," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, Sept 2012, pp. 469–476.

[88] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *STACS*, 1995, pp. 229–242. [Online]. Available: http://dx.doi.org/10.1007/3-540-59042-0_76

[89] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. G. Larsen, and D. Lime, "UPPAAL-Tiga: Time for playing games!" in *Proc. Int. Conf. Comput.-Aided Verification*. Springer, 2007, pp. 121–125.

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

[91] 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, July 2000.

[92] 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," *International Journal of Control*, vol. 76, no. 12, pp. 1171–1200, Aug. 2003.

[93] H. Wong-Toi, "The synthesis of controllers for linear hybrid automata," in *Decision and Control, 1997., Proceedings of the 36th IEEE Conference on*, vol. 5, Dec 1997, pp. 4607–4612 vol.5.

[94] 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. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2012.10.042

[95] D. Bresolin, L. Di Guglielmo, L. Geretti, and T. Villa, "Correct-by-construction code generation from hybrid automata specification," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, July 2011, pp. 1660–1665.

[96] L. Di Guglielmo, S. Seshia, and T. Villa, "Synthesis of implementable control strategies for lazy linear hybrid automata," in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, Sept 2013, pp. 1381–1388.

[97] M. D. Wulf, "From timed models to timed implementations," Ph.D. dissertation, Universite Libre de Bruxelles, 2006-7.

[98] M. De Wulf, L. Doyen, and J.-F. Raskin, "Almost ASAP semantics: From timed models to timed implementations," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, R. Alur and G. Pappas, Eds. Springer Berlin Heidelberg, 2004, vol. 2993, pp. 296–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24743-2_20

[99] M. Agrawal and P. Thiagarajan, "The discrete time behavior of lazy linear hybrid automata," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Berlin Heidelberg, 2005, vol. 3414, pp. 55–69. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31954-2_4

[100] 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, Automation and Test in Europe*, Mar. 2014, pp. 1–4.

[101] I. Moir and A. Seabridge, *Aircraft Systems: Mechanical, Electrical and Avionics Subsystems Integration. Third Edition*. Chichester, England: John Wiley and Sons, Ltd, 2008.

[102] (2012, Feb.) IBM ILOG CPLEX Optimizer. [Online]. Available: www.ibm.com/software/integration/optimization/cplex-optimizer/

[103] P. Nuzzo and A. Sangiovanni-Vincentelli, "Robustness in analog systems: Design techniques, methodologies and tools," in *Proc. IEEE Symp. Industrial Embedded Systems*, Jun. 2011.

[104] 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*, ser. Lecture Notes in Computer Science, J. Esparza and R. Majumdar, Eds. Springer Berlin Heidelberg, 2010, vol. 6015, pp. 23–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_3

[105] B. Caillaud, B. Delahaye, K. Larsen, A. Legay, M. Pedersen, and A. Wasowski, "Compositional design methodology with Constraint Markov Chains," in *Quantitative Evaluation of Systems (QEST), 2010 Seventh International Conference on the*, Sept 2010, pp. 123–132.

[106] G. Gössler, D. N. Xu, and A. Girault, "Probabilistic contracts for component-based design," *Formal Methods in System Design*, vol. 41, no. 2, pp. 211–231, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10703-012-0162-4

[107] 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)*, ser. LNCS (Tutorial Volume), M. Bernardo and J. Hillston, Eds., vol. 4486. Springer, 2007, pp. 220–270.

[108] ——, “PRISM 4.0: Verification of probabilistic real-time systems,” in *Proc. Int. Conf. Comput.-Aided Verification*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.