Title
Let’s Get Physical: Computer Science Meets Systems

ISBN
978-3-642-54847-5

Authors
Nuzzo, Pierluigi
Sangiovanni-Vincentelli, Alberto

Publication Date
2014

DOI
10.1007/978-3-642-54848-2_13

Peer reviewed

Let's get physical: computer science meets systems

Pierluigi Nuzzo and Alberto Sangiovanni-Vincentelli

University of California at Berkeley, Berkeley CA 94720, USA
nuzzo@eecs.berkeley.edu,alberto@eecs.berkeley.edu

Abstract. In cyber-physical systems (CPS) computing, networking and control (typically regarded as the "cyber" part of the system) are tightly intertwined with mechanical, electrical, thermal, chemical or biological processes (the "physical" part). The increasing sophistication and heterogeneity of these systems requires radical changes in the way sense-and-control platforms are designed to regulate them. In this paper, we highlight some of the design challenges due to the complexity and heterogeneity of CPS. We argue that such challenges can be addressed by leveraging concepts that have been instrumental in fostering electronic design automation while dealing with complexity in VLSI system design. Based on these concepts, we introduce a design methodology whereby platform-based design is combined with assume-guarantee contracts to formalize the design process and enable realization of CPS architectures and control software in a hierarchical and compositional manner. We demonstrate our approach on a prototype design of an aircraft electric power system.

Keywords: Cyber-physical systems, embedded systems, VLSI systems, electronic design automation, platform-based design, contract-based design, assume-guarantee contracts, aircraft electric power system.

1 Emerging Information Technology Trends

The emerging information technology scenario features a large number of new applications which go beyond the traditional "compute" or "communicate" functions. The majority of these applications build on *distributed sense and control systems* destined to run on highly heterogeneous platforms, combining large, high-performance compute clusters (the infrastructure core or "cloud") with broad classes of mobiles, in turn surrounded by even larger swarms of microscopic sensors [16]. Such *cyber-physical systems* (CPS) [19, 8, 6] are characterized by the tight integration of computation with mechanical, electrical, and chemical processes: networks monitor and control the physical processes, usually with feedback loops where physics affects computation and vice versa.

CPS have the potential to radically influence how we deal with a broad range of crucial problems facing our society today, from national security and safety, to energy management, efficient transportation, and affordable health

care. However, CPS complexity and heterogeneity, originating from combining what in the past have been separate worlds, tend to substantially increase *system* design and verification challenges. 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, Boeing's Airbus delay bringing an approximate toll of $6.6 billion are examples of devastating effects that design problems may cause. If this is the present situation, the problem of designing planetary-scale swarm systems appears insurmountable unless bold steps are taken to advance significantly the *science of design*.

While in traditional embedded system design the physical system is regarded as a given, the emphasis of CPS design is instead on managing dynamics, time, and concurrency by *orchestrating* networked, distributed computational resources *together* with the physical systems. Functionality in CPS is provided by an ensemble of sensing, actuation, connectivity, computation, storage and energy. Therefore, CPS design entails the convergence of several sub-disciplines, ranging from computer science, which mostly deals with computational aspects and carefully abstracts the physical world, to automatic control, electrical and mechanical engineering, which directly deals with the physical quantities involved in the design process. The inability to rigorously model the interactions among heterogeneous components and between the "physical" and the "cyber" sides is a serious obstacle to the efficient realization of CPS. 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.

In this paper, we highlight the main *design challenges* for the realization of embedded systems caused by the complexity and heterogeneity of CPS. Resting on the successful achievements of electronic design automation (EDA) in taming design complexity of VLSI systems [15], we argue that such challenges can only be addressed by employing *structured and formal design methodologies* that seamlessly and coherently combine the various dimensions of the multi-scale design space and provide the appropriate abstractions. We then introduce and demonstrate a CPS design methodology by combining the platform-based design (PBD) [17] and contract-based design (CBD) [16] paradigms.

2 Cyber-Physical System Design Challenges

In this section we highlight the main CPS design challenges, based on the elaborations in [6] and [16]. In particular, we categorize them in terms of modeling, integration and specification challenges.

2.1 Modeling Challenges

Model-based design (MBD) [20, 18] is today generally accepted as a key enabler for the design and integration of complex systems. However, CPS tend to stress all existing modeling languages and frameworks. 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, expressed as differential equations, which are acausal, concurrent models. Therefore, most of the modeling challenges stem by the difficulty in accurately capturing the interactions between these two worlds.

Challenge 1—Modeling Timing and Concurrency. A first set of technical challenges in analysis and design of real-time embedded software stems from the need to bridge its inherently sequential semantics with the intrinsically concurrent physical world. All the general-purpose computation and networking abstractions are built on the premise that execution time is just an issue of performance, not correctness. Therefore, timing of programs is not repeatable, except at very coarse granularity, and programmers have hard time to specify timing behaviors within the current programming abstractions. Moreover, concurrency, is often poorly modelled. Concurrent software is today dominated by threads, performing sequential computations with shared memory. Incomprehensible interactions between threads can be the sources of many problems, ranging from deadlock and scheduling anomalies, to timing variability, nondeterminism, buffer overruns, and system crashes. Finally, modeling distributed systems adds to the complexity of CPS modeling by introducing issues such as disparities in measurements of time, network delays, imperfect communication, consistency of views of system state, and distributed consensus [6].

Challenge 2—Modeling Interactions of Functionality and Implementation. To evaluate a CPS model, it is necessary to model the dynamics of software and networks. In fact, computation and communication do take time. However, pure functional models implicitly assume that data are computed and transmitted in zero time, so that the dynamics of the software and networks have no effect on system behavior. It is then essential to provide a mechanism to capture the interactions of functionality and implementation. Implementation is largely orthogonal to functionality and should therefore not be an integral part of a model of functionality. Instead, it should be possible to conjoin a functional model with an implementation model. The latter allows for design space exploration, while the former supports the design of control strategies. The conjoined models enable evaluation of interactions across these domains.

2.2 Integration Challenges

CPS integrate diverse subsystems, by often composing pieces that have been pre-designed or designed independently by different groups or companies. This is done routinely, for example, in the avionics and automotive sectors, albeit in a

heuristic and ad hoc way. Yet, integrating component models to develop holistic views of the system becomes very challenging, as summarized below.

Challenge 3—Keeping Model Components Consistent. Inconsistency may arise when a simpler (more abstract) model evolves into a more complex (refined) one, where a single component in the simple model becomes multiple components in the complex one. Moreover, non-functional aspects such as performance, timing, power or safety analysis are typically addressed in dedicated tools using specific models, which are often evolved independently of the functional ones (capturing the component dynamics), thus also increasing the risk of inconsistency. In a modeling environment, a mechanism for maintaining model consistency allows components to be copied and reused in various parts of the model while guaranteeing that, if later a change in one instance of the component becomes necessary, the same change is applied to all other instances that were used in the design. Additionally, such a mechanisms is instrumental in maintaining consistency between the results of specialized analysis and synthesis tools using different representations of the same component.

Challenge 4—Preventing Misconnected Model Components. The bigger a model becomes, the harder it is to check for correctness of connections between components. Typically model components are highly interconnected and the possibility of errors increases. Errors may be due to different units between a transmitting and a receiving port (unit errors), different interpretation of the exchanged data (semantic errors), or just reversed connections among ports (transposition errors). Since none of these errors would be detected by a type system, specific measures should be enabled to automatically check for them [6].

Challenge 5—Improving Scalability and Accuracy of Model Analysis Techniques. Conventional verification and validation techniques do not scale to highly complex or adaptable systems (i.e., those with large or infinite numbers of possible states or configurations). Simulation techniques may also be affected by modeling artifacts, such as solver-dependent, nondeterminate, or Zeno behaviors [6]. In fact, CPS may be modeled as hybrid systems integrating solvers that numerically approximate the solutions to differential equations with discrete models, such as state machines, dataflow models, synchronous-reactive models, or discrete event models. Then, when a threshold must be detected, the behavior defined by a model may depend on the selected step size, which is dynamically adjusted by the numerical solver to increment time.

2.3 Specification Challenges

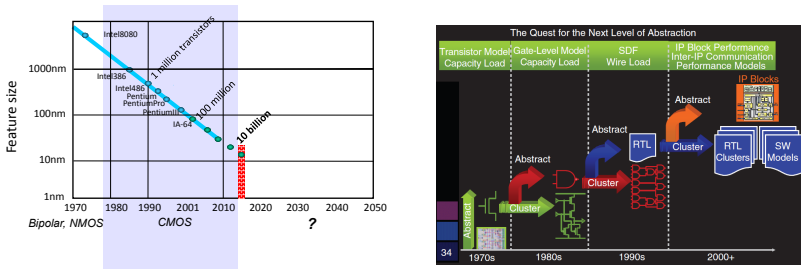
Depending on application domains, up to 50% of all errors result from imprecise, incomplete, or inconsistent and thus unfeasible requirements. The overall system product specification is somewhat of an art today, since to verify its completeness and its correctness there is little that it can be used to compare with.

Challenge 6—Capturing System Requirements. Among the many approaches taken in industry for getting requirements right, some of them are meant for initial systems requirements, mostly relying on ISO 26262 compliant approaches. To cope with the inherently unstructured problem of (in)completeness of requirements, industry has set up domain- and application-class specific methodologies. As particular examples, we mention learning processes, such as the one employed by Airbus to incorporate the knowledge base of external hazards from flight incidents, and the Code of Practice proposed by the Prevent Project, using guiding questions to assess the completeness of requirements in the concept phase of the development of advanced driver assistance systems. Use-case analysis methods as advocated for UML based development process follow the same objective. A common theme of these approaches is the intent to systematically identify those aspects of the environment of the system under development whose observability is necessary and sufficient to achieve the system requirements. However, the most efficient way of assessing completeness of a set of requirements is by executing it, which is only possible if semi-formal or formal specification languages are used, where the particular shape of such formalizations is domain dependent.

Challenge 7—Managing Requirements. Design specifications tend to move from one company (or one division) to the next in non-executable and often unstable and imprecise forms, thus yielding misinterpretations and consequent design errors. In addition, errors are often caught only at the final integration step as the specifications were incomplete and imprecise; further, nonfunctional specifications (e.g., timing, power consumption, size) are difficult to trace. It is common practice to structure system level requirements into several "chapters", "aspects", or "viewpoints", quite often developed by different teams using different skills, frameworks, and tools. Without a clean approach to handle multiple viewpoints, the common practice today is to discard some of the viewpoints in a first stage, e.g., by considering only functions and safety. Designs are then developed based on these only viewpoints. Other viewpoints are subsequently taken into account (e.g., timing, energy), thus resulting in late and costly modifications and re-designs.

3 Coping with Complexity in VLSI Design: Lessons Learned

Over the past decades, a major driver for silicon microelectronics research has been Moore's law, which conjectures the continued shrinkage of critical chip dimensions (see Fig. 1 (a)). Microelectronic progress became so predictable that the Semiconductor Industry Association (SIA) developed a road-map to help defining critical steps and sustaining progress; the material science and material processing research community has successfully met the challenges, maintaining a steady stream of results supporting continued scaling of CMOS devices to smaller dimensions. By taking full advantage of the availability of billion-transistor chips, increasingly higher performance Systems-on-Chip (SoC) are



(a) (b)

Fig. 1. (a) Reduction of minimum feature size with time in VLSI systems and (b) Levels of abstraction in VLSI design (Figure from [15])

being fabricated today, thus enabling new architectural approaches to information processing and communication. The appearance of new nano-scale devices is expected to revolutionize the way information is processed on chip, and perhaps more significantly, have a major impact on emerging applications at the intersection of the biological, information and micro-electro-mechanical worlds.

3.1 Dealing with Moore's Law

Dealing with steady increase in complexity over the decades has been made possible only because of the continuous increase of productivity brought by electronic design automation (EDA).

By looking back at the history of design methods, we can infer how the changes in design productivity have always been associated with a *rise in the level of abstraction of design capture*. As can be seen in Fig. 1 (b), in 1971, the highest level of abstraction for digital integrated circuits (IC) was the schematic of a transistor; ten years later, it became the digital gate; by 1990, the use of hardware description languages (HDL) was pervasive, and design capture was done at the register transfer level (RTL). Dealing with blocks of much coarser granularity than in the past has become essential in order to cope with the productivity increase the industry is asked to provide. The recent emphasis on SoC, bringing system-level issues into chip design, is a witness to this trend.

One of these issues relates to the concept of *system decomposition* and *integration out of pre-designed intellectual property (IP) blocks*. Although top-down decomposition has been customarily adopted by the semiconductor industry for years, it presents some limitations as a designer or a group of designers has to fully comprehend the entire system and to partition appropriately its various parts, a difficult task given the enormous complexity of today's systems. As mentioned in Section 2.2, an alternative is to develop systems by composing pre-designed pieces, whereby preserving compositionality is essential: the building blocks should be designed so that their properties are maintained when connected together to allow reuse without the need for expensive verification steps.

Decomposition and abstraction have been two basic approaches traditionally used to manage the design complexity. However, complexity has been also managed *by "construction"*, i.e. by "artificially" constraining the space to regular, or modular design styles that can ease design verification (e.g. by enforcing regular layout and synchronous design), and by *structured methodologies*, which start high in the abstraction layers and define a number of refinement steps that go from the initial description to the final implementation.

The design problems faced in SoC design are very similar to the ones discussed in Section 2, the main difference between them being the importance given to time-to-market and to the customer appeal of the products versus safety and hard-time constraints. Several languages and design tools have been proposed over the years to enable checking system level properties or explore alternative architectural solutions for the same set of requirements. Among others, we recall generic modeling frameworks, such as Matlab/Simulink[1] or Ptolemy II[2], hardware description languages, such as Verilog[3] or VHDL[4], transaction-level modeling tools, such as SystemC[5], together with their respective analog-mixed-signal extensions[6], modeling languages for architecture modeling, such as SysML[7] and AADL[8]. Some of these tools focus on simulation while others are geared towards performance modeling, analysis and verification. However, the design technology challenge is to address the entire system design process and not to consider only point solutions of methodology, tools, and models that ease part of the design. This calls for new modeling approaches that can mix different physical systems, control logic, and implementation architectures. In doing so, existing approaches, models, and tools should be subsumed and not eliminated in order to be smoothly incorporated in current design flows. A design platform should then be developed to host the new techniques and to integrate a set of today's poorly interconnected tools.

3.2 System Design Methodology

The considerations above motivate the view that a unified methodology and framework could be used across several different industrial domains. Among the lines of attack developed by research institutions and industry to cope with the exponential complexity growth, a design paradigm of particular interest to the development of embedded systems is the V-model, a widely accepted scheme in the defense and transportation domains.

---

[1] http://www.mathworks.com/products/simulink
[2] http://ptolemy.eecs.berkeley.edu
[3] http://www.verilog.com/
[4] http://www.vhdl.org
[5] http://www.accellera.org/downloads/standards/systemc
[6] http://www.eda.org/verilog-ams/
http://www.eda.org/vhdl-ams/
http://www.accellera.org/downloads/standards/systemc/ams
[7] http://www.omg.org/spec/SysML
[8] http://www.aadl.info/aadl/currentsite

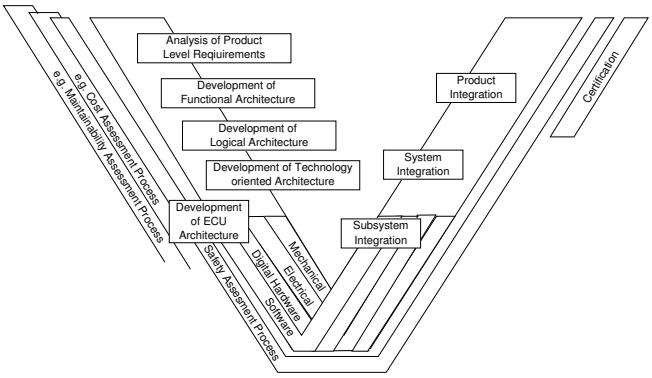


Fig. 2. The V Model.

The *V-model* was originally developed for defense applications by the German DoD.[9] It structures the product development processes into a *design* and an *integration* phase along variations of the V diagram shown in Fig. 2. Specifically, following product level requirement analysis, subsequent steps would first evolve a functional architecture supporting product level requirements. Sub-functions are then re-grouped taking into account re-use and product line requirements into a logical architecture, whose modules can be developed independently, e.g., by different subsystem suppliers. The realization of such modules often involves mechatronic design. The top-level of the technology-oriented architecture would then show the mechatronic architecture of the module, defining interfaces between the different domains of mechanical, hydraulic, electrical, and electronic system design. Subsequent phases would then unfold the detailed design for each of these domains, such as the design of the electronic subsystem involving among others the design of electronic control units. These design phases are paralleled by integration phases along the right-hand part of the V, such as integrating basic and application software on the ECU hardware to actually construct the electronic control unit, integrating the complete electronic subsystems, integrating the mechatronic subsystem to build the module, and integrating multiple modules to build the complete product. An integral part of V-based development processes are testing activities, where at each integration level test-suites developed during the design phases are used to verify compliance of the integrated entity to their specification. Since system integration and validation may often be performed too late in the design flow, there is limited ability to predict, early in the design process, the consequences of a design decision on system performance and the cost of radical departures from known designs. Therefore, design-space exploration is rarely performed adequately, yielding suboptimal designs where the architecture selection phase does not consider extensibility, re-usability, and

[9] `http://www.v-model-xt.de`

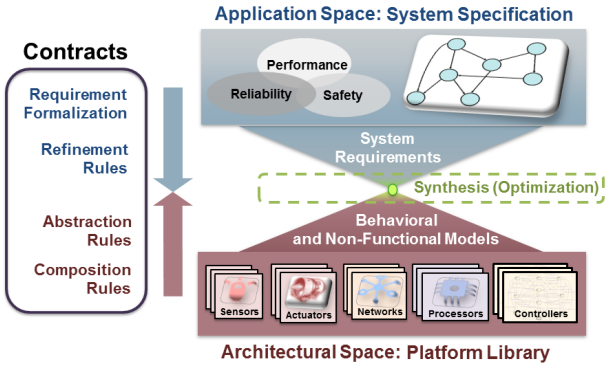


Fig. 3. Platform-based design and the role of contracts.

fault tolerance to the extent that is needed to reduce cost, failure rates, and time-to-market.

Platform-based design (PBD) was introduced in the late 1980s to capture a design process that could encompass horizontal and vertical decompositions, and multiple viewpoints and in doing so, support the supply chain as well as multi-layer optimization [17].

Platform-based design addresses already many of the challenges outlined in Section 2. Its concepts have been applied to a variety of very different domains: from automotive, to System-on-Chip, from analog circuit design, to building automation to synthetic biology. By limiting the design space to the platform library, it allows efficient design exploration and optimization, aiming to correct-by-construction solutions (Challenge 5). Moreover, the meet-in-the-middle approach, where functional models are combined with non-functional models, and successive top-down refinements of high-level specifications across design layers are mapped onto bottom-up abstractions and characterizations of potential implementations, allows effectively coupling system functionality and architecture (Challenge 1, 2 and 5), as represented in Fig. 3. However, to successfully deploy such a methodology, we need rigorous mechanisms for (i) determining *valid* compositions of compatible components so that when the design space is explored, only *legal* compositions are taken into consideration; (ii) guaranteeing that a component at a higher level of *abstraction* is an accurate representation of a lower level component (or aggregation of components); (iii) checking that an architecture platform is indeed a correct *refinement* of a specification platform, and (iv) formalizing top-level system *requirements*. In the following section, we show how such goals can be achieved by combining platform-based design with the concept of *contracts*.

4 Platform-Based Design With Contracts

The notion of contracts originates in the context of assume-guarantee reasoning. Informally, a contract is a pair $\mathcal{C} = (A, G)$ of properties, assumptions and guarantees, respectively representing the assumptions on the environment and the

promises of the system under these assumptions. The essence of contracts is a compositional approach, where 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.

Compositional reasoning has been known for a long time, but it has mostly been used as a verification mean for the design of software. Rigorous contract theories have then been developed over the years, including assume-guarantee (A/G) contracts [4] and interface theories [1]. However, their concrete adoption in CPS design is still at its infancy. Examples of application of A/G contracts have only been recently demonstrated in the automotive [5] and consumer electronics [12] domains. The use of A/G contracts for control design in combination with PBD has been advocated in [16], while in [13, 11], a PBD methodology is first introduced that uses contracts to integrate heterogeneous modeling and analysis frameworks for synthesis and optimization of CPS architectures and control protocols. The design flow is demonstrated on a real-life example of industrial interest, namely the design of system topology and supervisory control for aircraft electric power systems (EPS).

4.1 Contracts

We summarize the main concepts behind our methodology by presenting a simple generic contract model centered around the notion of platform *component*. A platform component $\mathcal{M}$ can be seen as an abstraction representing an element of a design, characterized by a set of *attributes*, including: *variables* (input, output and internal), configuration *parameters*, and *ports* (input, output and bidirectional); a *behavioral model*, uniquely determining the values of the output and internal variables given the values of the input variables and configuration parameters, and 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. Components can be connected together by sharing certain ports under constraints on the values of certain variables. In what follows, we use variables to denote both component variables and ports. A component may be associated with both implementations and contracts. An *implementation* M is an instantiation of a component $\mathcal{M}$ for a given set of configuration parameters. In the following, we also use M to denote the set of behaviors of an implementation, which assign a history of "values" to ports. Behaviors are generic and abstract. For instance, they could be continuous functions that result from solving differential equations, or sequences of values or events recognized by an automata model.

A *contract* $\mathcal{C}$ for a component $\mathcal{M}$ is a pair of assertions (A, G), called the *assumptions* and the *guarantees*, each representing a specific set of behaviors over the component variables [4]. An implementation M satisfies an assertion B whenever M and B are defined over the same set of variables and all the behaviors of M satisfy the assertion, i.e. when $M \subseteq B$. An implementation of a component satisfies a contract whenever it satisfies its guarantee, subject to

the assumption. Formally, $M \cap A \subseteq G$, where M and $\mathcal{C}$ have the same variables. We denote such a *satisfaction* relation by writing $M \models \mathcal{C}$. An implementation E is a legal *environment* for $\mathcal{C}$, i.e. $E \models_E \mathcal{C}$, whenever $E \subseteq A$. Two contracts $\mathcal{C}$ and $\mathcal{C}'$ with identical variables, identical assumptions, and such that $G' \cup \neg A = G \cup \neg A$, where $\neg A$ is the complement of A, possess identical sets of environments and implementations. Such two contracts are then *equivalent*. In particular, any contract $\mathcal{C} = (A, G)$ is equivalent to a contract in *saturated form* (A, G'), obtained by taking $G' = G \cup \neg A$. Therefore, in what follows, we assume that all contracts are in saturated form. 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 $G \neq \emptyset$. Let M be any implementation, i.e. $M \models \mathcal{C}$, then $\mathcal{C}$ is *compatible*, if there exists a legal environment E for M, 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.

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 M_1 and M_2 two components that are composable to obtain $M_1 \times M_2$ and satisfy, respectively, contracts $\mathcal{C}_1$ and $\mathcal{C}_2$. Then, $M_1 \times M_2$ is a valid composition if M_1 and M_2 are *compatible*. This can be checked by first computing the contract composition $\mathcal{C}_{12} = \mathcal{C}_1 \otimes \mathcal{C}_2$ and then checking whether $\mathcal{C}_{12}$ is compatible. To compose multiple views of the same component that need to be satisfied simultaneously, the *conjunction* ($\wedge$) of contracts can also be defined so that if $M \models \mathcal{C}_1 \wedge \mathcal{C}_2$, then $M \models \mathcal{C}_1$ and $M \models \mathcal{C}_2$. Contract conjunction can be computed by defining a preorder on contracts, which formalizes a notion of *refinement*. We say that $\mathcal{C}$ refines $\mathcal{C}'$, written $\mathcal{C} \preceq \mathcal{C}'$ 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 $E \models_E \mathcal{C}$. Mathematical expressions for computing contract composition and conjunction can be found in [4].

Horizontal and Vertical Contracts. Since compatibility is assessed among components at the same abstraction layer, the first category of contracts presented above can be denoted as *horizontal contracts*. On the other hand, *vertical contracts* can also be used to verify whether the system obtained by composing the library elements according to the horizontal contracts satisfies the requirements posed at the higher level of abstraction. If these sets of contracts are satisfied, the mapping mechanism of PBD can be used to produce design refinements that are correct by construction. Vertical contracts are tightly linked to the notions of mapping of an application onto an implementation platform [12]. However, compositional techniques that check correct refinement on each subsystem independently are not effective, in general, since the specification architecture at level $l+1$ may be defined in an independent way, and does not generally match the implementation architecture at level l. Let $\mathcal{S} = \otimes_{i \in I} \mathcal{S}_i$ and $\mathcal{A} = \otimes_{j \in J} \mathcal{A}_j$ be the two contracts describing the specification and implementation platforms, re-

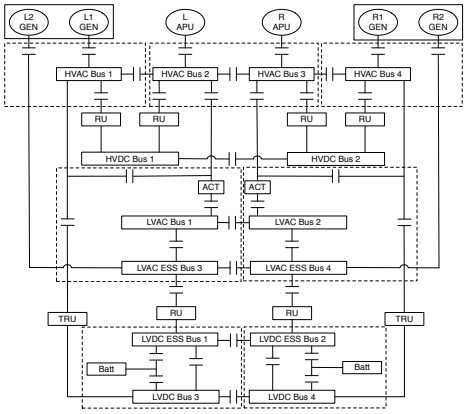


Fig. 4. Single-line diagram of an aircraft electric power system (Figure from [13]).

spectively. Therefore, verification of vertical contracts can be performed through mapping of the application to the implementation platform as follows. In this example, they are compositionally defined out of I and J components, which may not directly match. Then, the mapping of the specification over the implementation can be modelled by the composition $\mathcal{S} \otimes \mathcal{A}$, and checking vertical contracts becomes equivalent to checking that $\mathcal{S} \otimes \mathcal{A}$ refines $\mathcal{S}$, which can be performed compositionally.

4.2 A Contract-Based Design Flow for CPS

We now show how the challenges discussed in Section 2 can effectively be addressed by a platform-based design methodology using contracts. As an example, we consider the embedded control design problem in [13]. Fig. 4 shows a sample structure of an aircraft EPS in the form of a single-line diagram, a simplified notation for three-phase power systems. Generators deliver power to the loads via buses. Essential loads cannot be unpowered for more than a predefined period t_{max}, i.e. a typical specification would require that the failure probability for an essential load (i.e., the probability of being unpowered for longer than t_{max}) be smaller than 10^{-9} per flight hour. Contactors are electromechanical switches that are opened or closed to determine the power flow from sources to loads. The goal is to design the system topology (e.g. number and interconnection of components) and the controller to accommodate all changes in system conditions or failures, and reroute power by appropriately actuating the contactors, so that essential buses are adequately powered.

In our design flow, pictorially represented in Fig. 5, platform component design and characterization is completely orthogonalized from system specification and algorithm design.

Platform Library Generation. In the bottom-up phase of the design process, a library of components (and contracts) is generated to model (or specify) both

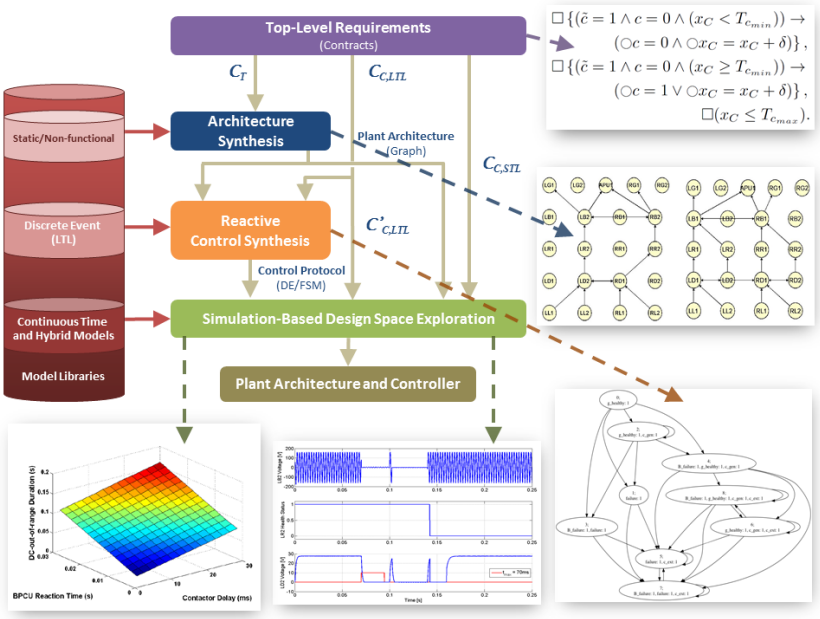


Fig. 5. Contract-based CPS design flow and its demonstration to an aircraft electrical power system [13].

the plant architecture (e.g. the power system topology in Fig. 5) and the controller. Components can be hierarchically organized to represent the system at different levels of abstraction, e.g. *steady-state* (static), *discrete-event* (DE), and *hybrid* levels. At each level of abstraction, components are also capable of exposing multiple, complementary *views*, associated with different design concerns (e.g. safety, performance, reliability) and with models that can be expressed via different formalisms (e.g. graphs, linear temporal logic, differential equations in Fig. 5), and analyzed by different tools. Such models include non-functional (performance) metrics, such as timing, energy and cost. Contracts allow checking consistency among models as the library evolves, which addresses Challenge 3.

Requirement Formalization. In the top-down phase of the design process, top-level system requirements are formalized as contracts. Responsibilities of achieving requirements are split into those to be established by the system (guarantees) and those characterizing admissible environments (assumptions). As an example, controller requirements can be expressed as a contract $\mathcal{C}_C = (A_C, G_C)$, where A_C encodes the allowable behaviors of the environment (physical plant) and G_C encodes the top-level system requirements. To define $\mathcal{C}_C$, formal specification languages can be used, e.g. linear temporal logic (LTL) [14] and signal temporal logic (STL) [9] in Fig. 5, to allow reasoning about temporal aspects of systems at

different levels of abstraction. Using contracts resting on logic-based formalisms comes with the advantage that "spurious" unwanted behaviors can be excluded by "throwing in" additional contracts, or strengthening assumptions, or by considering additional cases for guarantees, thus addressing Challenge 6. A second advantage rests in the capability of checking for consistency by providing effective tests, whether a set of contracts is realizable, or whether, in contrast, facets of these are inherently conflicting, and thus no implementation is feasible, which addresses Challenge 7. By reflecting the model library, the particular shape of requirement formalizations is also viewpoint and domain dependent. To address Challenge 1, such system-level models should still come with a rigorous temporal semantics that allows specifying the interaction between the control program and the physical dynamics so as to model timing and concurrency at a higher abstraction level in a way that is largely independent of underlying hardware details. For instance, LTL allows reasoning about the temporal behaviors of systems characterized by Boolean, discrete-time signals or sequences of events (DE abstraction). On the other hand, STL deals with dense-time real signals and hybrid dynamical models that mix the discrete dynamics of the controller with the continuous dynamics of the plant (hybrid abstraction).

Mapping Functions to Implementations. By leveraging models expressed in different formalisms, the design is cast as a set of problems mapping functions over implementations. The mapping problem is a synthesis problem that can be solved by either leveraging pre-existing synthesis tools, or by casting an optimization problem that uses information from both the system and the component levels to evaluate global tradeoffs among components.

In the example of Fig. 5, $\mathcal{C}_T$ is first used together with steady-state models of the plant components and a template of the EPS topology (represented as a graph) to synthesize a final topology that minimize component cost subject to connectivity, power flow and reliability constraints, all expressed as mixed integer-linear constraints. $\mathcal{C}_{C,LTL}$ is then used together with DE models of the plant components (also described by LTL formulas) and the EPS topology, to synthesize a reactive control protocol in the form of one (or more) state machines. Reactive synthesis tools [10, 7] can be used to generate control logic from LTL A/G contracts. The resulting controller will then satisfy $\mathcal{C}_{C,LTL}$ by construction. Satisfaction of $\mathcal{C}_{C,STL}$ is then assessed on a hybrid model, including both the controller and an acausal, equation-based representation of the plant, by monitoring simulation traces while optimizing a set of system parameters. Contracts are captured as optimization constraints. The resulting optimal controller configuration is returned as the final design.

Horizontal contracts allow checking or enforcing compatibility and correct connections among components, thus addressing Challenge 4. Vertical contracts allow checking or enforcing correct abstraction and refinement relations, thus maintaining consistency among platform instances, models and requirements at different abstraction levels (Challenge 4 and 7). Moreover, in control design, vertical contracts define relations between the properties of the controller and the ones of its execution platform, which helps address Challenge 2. Typically,

the controller defines requirements in terms of several aspects that include 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.

The association of functionality to architectural services to evaluate the characteristics (such as latency, throughput, power, and energy) of a particular implementation by simulation (Challenge 2) can be supported by frameworks such as Metropolis [2, 3], which is founded on design representation mechanisms that can be shared across different models of computation and different layers of abstraction. A typical design scenario would then entail a front-end *orchestrator* routine responsible for coordinating a set of back-end specialized synthesis and optimization frameworks, each dealing with a different representation of the platform, and consistently process their results. To maintain such consistency and improve on the scalability of the specific synthesis and optimization algorithms (Challenge 5), such an orchestrator should 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.

5 Conclusions

Dealing with the heterogeneity and complexity of cyber-physical systems requires innovations in design technologies and tools. In this paper, we have advocated the need for a design and integration platform that can operate at different levels of abstraction, orchestrate hardware and software, digital and analog, cyber and physical subsystem design, as well as facilitate the integration of IP blocks and tools. Then, we have introduced a platform-based design methodology enriched with contracts and demonstrated its potential to provide the foundations for such a framework.

Acknowledgments

This work was supported in part by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium and 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.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. Symp. Foundations of Software Engineering. pp. 109–120. ACM Press (2001)

2. Balarin, F., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L., Watanabe, Y.: Metropolis: an integrated electronic system design environment. Computer 36(4) (2003)
3. Balarin, F., Davare, A., D'Angelo, M., Densmore, D., Meyerowitz, T., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A., Simalatsar, A., Watanabe, Y., Yang, G., Zhu, Q.: Platform-based design and frameworks: METROPOLIS and METRO II. In: Nicolescu, G., Mosterman, P.J. (eds.) Model-Based Design for Embedded Systems, chap. 10, p. 259. CRC Press, Taylor and Francis Group, Boca Raton, London, New York (November 2009)
4. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Formal methods for components and objects. chap. Multiple Viewpoint Contract-Based Specification and Design, pp. 200–225. Springer-Verlag, Berlin, Heidelberg (2008)
5. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., et al.: Contracts for System Design. Rapport de recherche RR-8147, INRIA (Nov 2012)
6. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.: Modeling cyber-physical systems. Proc. IEEE 100(1), 13–28 (January 2012)
7. Emerson, E.A.: Temporal and modal logic. Handbook of theoretical computer science 2, 995–1072 (1990)
8. Lee, E.A.: Cyber physical systems: Design challenges. In: Proc. IEEE Int. Symposium on Object Oriented Real-Time Distributed Computing. pp. 363–369 (May 2008)
9. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Modeling and Analysis of Timed Systems. pp. 152–166 (2004)
10. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems: Specification, vol. 1. springer (1992)
11. Nuzzo, P., Finn, J.B., Iannopollo, A., Sangiovanni-Vincentelli, A.L.: Contract-based design of control protocols for safety-critical cyber-physical systems. In: Proc. Design, Automation and Test in Europe (Mar 2014)
12. Nuzzo, P., Sangiovanni-Vincentelli, A., Sun, X., Puggelli, A.: Methodology for the design of analog integrated interfaces using contracts. IEEE Sensors J. 12(12), 3329–3345 (Dec 2012)
13. Nuzzo, P., Xu, H., Ozay, N., Finn, J., Sangiovanni-Vincentelli, A., Murray, R., Donze, A., Seshia, S.: A contract-based methodology for aircraft electric power system design. Access, IEEE 2, 1–25 (2014)
14. Pnueli, A.: The temporal logic of programs. In: Symp. Foundations of Computer Science. vol. 31, pp. 46–57 (Nov 1977)
15. Sangiovanni-Vincentelli, A.: Corsi e ricorsi: The EDA story. IEEE Solid State Circuits Magazine 2(3), 6–26 (2010)
16. Sangiovanni-Vincentelli, A., Damm, W., Passerone, R.: Taming Dr. Frankenstein: Contract–based design for cyber–physical systems. In: Conf. Decision and Control (Dec 2011)
17. Sangiovanni-Vincentelli, A.: Quo vadis, SLD? reasoning about the trends and challenges of system level design. Proceedings of the IEEE 95(3), 467–506 (2007)
18. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
19. Sztipanovits, J.: Composition of cyber-physical systems. In: Proc. IEEE Int. Conf. and Workshops on Engineering of Computer-Based Systems. pp. 3–6 (March 2007)
20. Sztipanovits, J., Karsai, G.: Model-integrated computing. IEEE Computer pp. 110–112 (1997)