Contract-Based Design of Control Protocols for Safety-Critical Cyber-Physical Systems

Pierluigi Nuzzo, John B. Finn, Antonio Iannopollo and Alberto L. Sangiovanni-Vincentelli
EECS Department, University of California at Berkeley, Berkeley, CA 94720
Email:{nuzzo, jbfinn, antonio, alberto}@eecs.berkeley.edu

***Abstract*— We introduce a platform-based design methodology that addresses the complexity and heterogeneity of cyber-physical systems by using assume-guarantee contracts to formalize the design process and enable realization of control protocols in a hierarchical and compositional manner. Given the architecture of the physical plant to be controlled, the design is carried out as a sequence of refinement steps from an initial specification to a final implementation, including synthesis from requirements and mapping of higher-level functional and non-functional models into a set of candidate solutions built out of a library of components at the lower level. Initial top-level requirements are captured as contracts and expressed using linear temporal logic (LTL) and signal temporal logic (STL) formulas to enable requirement analysis and early detection of inconsistencies. Requirements are then refined into a controller architecture by combining reactive synthesis steps from LTL specifications with simulation-based design space exploration steps. We demonstrate our approach on the design of embedded controllers for aircraft electric power distribution.**

I. Introduction

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 behaviors (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 [1]. In the absence of a comprehensive modeling formalism, a structured design methodology should be adopted that consistently integrates existing design techniques, modeling practices and tools to perform design-space exploration across different domains. Moreover, because of the safety-critical nature of most CPS applications, methods to formally and functionally assess system operation and performance from abstract models all the way down to embedded software deployment are highly desired.

A severe limitation in common system 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 simulations later in the design process and prototype tests. The inability to rigorously model the interactions among heterogeneous components and between the physical and the cyber sides of the system is also a serious obstacle. Thus, the traditional heuristic design process based on informal requirement capture, designers' experience, and the V-model [2] 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.

To overcome the limitations above, several languages and 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 or VHDL[3], transaction-level modeling tools, such as SystemC[4], and object oriented languages for architecture modeling, such as SysML[5] and AADL[6]. Some of these tools focus on simulation while others are geared towards performance modeling, analysis and verification. However, an all-encompassing framework that helps interconnecting existing modeling, analysis and optimization tools, each operating on a different system or component representation, is still missing.

This work was partially supported by IBM and United Technologies Corporation (UTC) via the iCyPhy consortium and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

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

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

[3] http://www.verilog.com/, http://www.vhdl.org/

[4] http://www.accellera.org/home/

978-3-9815370-2-4/DATE14/©2014 EDAA

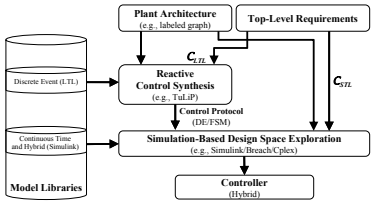


Fig. 1. Contract-based control design flow and tool chain.

In this paper, we introduce a methodology for the design of CPS embedded control protocols that addresses the above issues via a rigorous process founded on platform-based design (PBD) [3] and contracts [1]. The notion of contracts originates in the context of compositional assume-guarantee reasoning, which has been used for a long time, mostly as a verification mean for software design. Rigorous contract theories have then been developed over the years, including assume-guarantee (A/G) contracts [4] and interface theories [5]. However, their concrete adoption in CPS design is still at its infancy. The contribution of this paper is twofold: (i) we introduce a new methodology that uses contracts to integrate heterogeneous modeling and analysis frameworks for control system synthesis and optimization, in addition to verification, and (ii) we demonstrate it on a real-life example of industrial interest, namely supervisory control design for aircraft electric power systems (EPS). We illustrate the methodology and its application in Section II and Section III of the paper, respectively. Finally, we draw some conclusions in Section IV.

II. Contract-Based Design for CPS Control Protocols

We regard a *control system* as the composition of a physical *plant*, including sensors and actuators, and an embedded *controller* that runs a control protocol (logic) to restrict the behaviors of the plant so that all the remaining behaviors satisfy a set of system specifications. In particular, we focus on the design of *reactive controllers*, i.e. controllers that maintain an ongoing relation with their environment by appropriately reacting to it. Given a plant architecture, our goal is to find a controller that satisfies a set of top-level requirements or declare that no such controller exists.

Following the PBD paradigm, our design flow progresses in two precisely defined abstraction levels, namely *discrete-event* (DE) and *hybrid*, as shown in Fig. 1. At each level, 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 a *library* (collection) of *components* and *composition rules*. In the top-down phase of the design process, we formalize the requirements and associate them to different entities in the system. In the bottom-up phase, we build a library of components (and contracts) to model (or specify) both the plant architecture and the controller. We present the formal underpinnings of our methodology starting with the notion of components and contracts.

[5] http://www.omg.org/spec/SysML/

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

A. Components

A *component* $\mathcal{M}$ 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* $\lambda \in \Lambda$. 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 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;
- 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 *views*, 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. 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. A contract is a *specification* for $\mathcal{M}$, as detailed below.

B. Contracts

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]. Contract *refinement* checking can be used to verify that an architecture platform satisfying $\mathcal{C}_A$ is a correct implementation of a specification $\mathcal{C}_S$ at a higher level of abstraction. A special case of *heterogeneous refinement* occurs when $\mathcal{C}_A$ and $\mathcal{C}_S$ are expressed by using different formalisms. In this case, the behaviors expressed by one of the contracts must be mapped to the domain of the other contract via a *mapping* $\mathcal{M}$ before the refinement relation can be verified.

C. Requirement Formalization

Contracts play a key role in formalizing and analyzing component and system requirements. In particular, controller requirements can be expressed as a contract $\mathcal{C}_S = (A_S, G_S)$, where A_S encodes the allowable behaviors of the environment (physical plant) and G_S encodes the top-level system requirements. To define $\mathcal{C}_S$, we use two formal specification languages that allow reasoning about temporal aspects of systems at different levels of abstraction, namely linear temporal logic (LTL) [6] and signal temporal logic (STL) [7]. 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, we use STL to deal 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).

$\mathcal{C}_S$ 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, $\mathcal{C}_{LTL}$ 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. The resulting controller will then satisfy $\mathcal{C}_{LTL}$ by construction. Satisfaction of $\mathcal{C}_{STL}$ is then assessed on a hybrid model, generally including both the controller and an acausal, 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. In what follows, we provide the formulation for both the reactive synthesis and design space exploration steps in Fig. 1.

D. Reactive Control Synthesis

In the DE abstraction of LTL [8], a *system* is represented as a set $S_{DE} \in \mathcal{S}_{DE}$ of variables, where $\mathcal{S}_{DE}$ is the set of valuations of S_{DE}. LTL formulas are interpreted over infinite sequences of states, each state $s \in \mathcal{S}_{DE}$ being a valuation over S_{DE}. We call such sequences, of the form $\sigma = s_0 s_1 s_2 \ldots$, *behaviors* of the system.

Let E and D be sets of environment (input) and controlled (output) variables, respectively, of a controller (system) M_{DE}. 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. Then, the controller design problem reduces to a *reactive synthesis* problem [9], i.e. $M_{DE} \models \mathcal{C}_{LTL}$ if and only if $M_{DE} \models (\varphi_e \rightarrow \varphi_s)$. Reactive synthesis is concerned with constructing a control protocol (a partial function $f : (s_0 s_1 \ldots s_{t-1}, e_t) \mapsto d_t$) which chooses the move 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 protocol exists, the specification $\varphi = (\varphi_e \rightarrow \varphi_s)$ is said to be *realizable*. Any solution of the reactive synthesis problem is, therefore, an implementation of $\mathcal{C}_{LTL}$. If φ is unrealizable, then $\mathcal{C}_{LTL}$ is inconsistent. If φ_e is unsatisfiable, then $\mathcal{C}_{LTL}$ is incompatible.

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 [9]. Given a GR(1) specification, there is a set of tools that generate a finite-state automaton representing the control protocol for the system. For the example discussed in this paper, we used the Temporal Logic Planning (TuLiP) toolbox [10]. Such an automaton satisfies "by construction" the requirements in $\mathcal{C}_{LTL}$ and is provided as a condidate controller for the next exploration and optimization step.

E. Simulation-Based Design Space Exploration

At the hybrid abstraction level, a *signal* is a function mapping the time domain $\mathbb{T} = \mathbb{R}_{\geq 0}$ to the reals $\mathbb{R}$. A multi-dimensional signal $\boldsymbol{q}(t) = (q_1(t), \cdots, q_n(t))$ is a vector, where the i-th component $q_i(t)$ is a signal. The behavioral model $\mathcal{F}_H$ (e.g. implemented in a simulator) takes as input a signal $\boldsymbol{u}(t)$ and computes an output signal $\boldsymbol{y}(t)$ and an internal signal $\boldsymbol{x}(t)$ such that $\mathcal{F}_H(\boldsymbol{u}(t), \boldsymbol{y}(t), \boldsymbol{x}(t), \boldsymbol{\kappa}) = 0$, where $\boldsymbol{\kappa} \in \mathcal{K}$ is a vector of platform configuration parameters, i.e., a vector of variables that can be selected as a result of the design process. A collection of signals resulting from a simulation of the system is a *trace*, which can also be viewed as a multi-dimensional signal. Then, a system *behavior* is a trace $\boldsymbol{s}(t)$ that includes all the system input, output and internal signals.

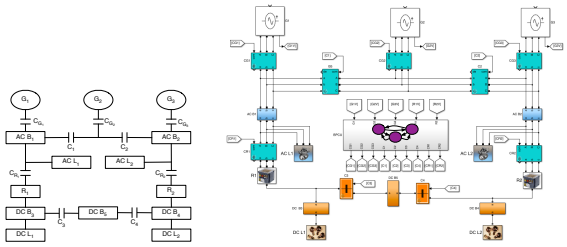
Let $\mathcal{C}_{STL} = (\phi_e, \phi_e \rightarrow \phi_s)$, with ϕ_e and ϕ_s parametric STL (PSTL) formulas, i.e. STL formulas where some numeric constants are replaced by symbolic parameters. Let $\boldsymbol{C}$ be an array of costs. The goal of design exploration is to find a set of configuration parameter vectors $\boldsymbol{\kappa}^*$ that are Pareto optimal with respect to the objectives in $\boldsymbol{C}$, while guaranteeing that the system satisfies ϕ_s for all possible traces $\boldsymbol{s} \in \mathcal{S}$ satisfying the environment assumptions ϕ_e. Examples of design parameters could be the controller clock or a tunable delay in a component. More formally, design exploration can be cast as a multi-objective robust optimization problem

$$\min_{\boldsymbol{\kappa} \in \mathcal{K}, \boldsymbol{\pi} \in \Pi} \quad \boldsymbol{C}(\boldsymbol{\kappa}, \boldsymbol{\pi}) \tag{1}$$
$$\text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{s}, \boldsymbol{\kappa}) = 0 \\ \\ \boldsymbol{s} \models \phi_s(\boldsymbol{\pi}) \qquad \forall \boldsymbol{s} \text{ s.t. } \boldsymbol{s} \models \phi_e(\boldsymbol{\pi}) \end{cases}$$

where $\boldsymbol{\pi}$ 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 $\boldsymbol{\kappa'}$, $\boldsymbol{s'} = (\boldsymbol{u'}, \boldsymbol{y'}, \boldsymbol{x'})$ is the trace of input, output and internal signals that are obtained by simulating $\mathcal{F}(.)$. We use the Breach toolbox [11] to facilitate post-processing of simulation traces and verify the satisfaction of STL formulas. A multi-objective optimization algorithm with simulation in the loop can then be implemented to find the Pareto optimal solutions $\boldsymbol{\kappa}^*$. While this may be expensive in general, it becomes affordable in many practical cases, as will be shown in Section III.

III. Aircraft Power Distribution Design Example

Fig. 2 (a) shows a sample structure of an aircraft EPS in the form of a single-line diagram, a simplified notation for three-phase power systems [12]. Generators (e.g., one on the left and one on the right side of the aircraft) deliver power to the loads (e.g. avionics, lighting, heating and motors) via AC and DC buses, while the Auxiliary Power Unit (APU) or batteries are used when one of the generators fails. Essential buses 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. Transformer Rectifier Units (TRUs) convert and rout AC power to DC buses. The goal of the controller, denoted as bus power control unit (BPCU), 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) (b)

Fig. 2. (a) Simplified single-line diagram of an aircraft power generation and distribution system and (b) its Simulink hybrid model representation.

A. Requirements

The DE representation of the controller M_{DE} includes as input variables the health statuses of generators, APU, and TRUs. Output variables are contactors, and can each take values of open (0) or closed (1). Under the assumption of a synchronous system, the timing view of M_{DE} includes a clock period or reaction time T_r as a configuration parameter. To capture system requirements as LTL formulas, we denote components by uppercase letters and component statuses (variables) by lowercase letters, and follow a similar approach as in [13]. Our requirement set includes 6 LTL formulas to capture the environment assumptions, 14 LTL formulas to capture the system guarantees in $\mathcal{C}_{LTL}$, and 2 STL formulas for $\mathcal{C}_{STL}$. Representative examples of assumptions and guarantees are provided below.

A_1. Reliability level: The overall reliability level of the system r is the probability that an essential bus cannot be powered by any of the available generators. Assuming independence of component failures, r determines the environment assumptions for the controller by providing a bound on the number of component failures allowed. For instance, by assuming that only generators and rectifiers can fail with probabilities $p_G = 2 \cdot 10^{-6}$ and $p_R = 10^{-5}$, respectively, our topology provides an overall reliability level of 10^{-10}. An environment assumption can then be specified as $\square\{(g_1 \vee g_2 \vee g_3) \wedge (r_1 \vee r_2)\}$, stating that no more than two generator and one rectifier unit may be unhealthy at any given time.

A_2. Irreversible failures: As a second environment assumption, we require that when a component fails during the flight, it will not come back online. This can be expressed in LTL as $\square \bigwedge_{i \in \mathcal{I}} \{\neg e_i \rightarrow \bigcirc(\neg e_i)\}$, where e_i is an environment variable and $\mathcal{I}$ is an index set enumerating the set of environment variables.

G_1. Unhealthy sources: As an example of LTL guarantees, we require that the set of contactors directly connected to an unhealthy source E_i be open to isolate it from the rest of the system. For our topology, we enforce $\square \bigwedge_{i \in \mathcal{I}} (\neg e_i \rightarrow \neg c_{E_i})$.

G_2. Controller reaction time: A DC essential bus can be unpowered for no longer than t_{max} in case of failure. To capture such timing requirement by accounting for plant and controller delays, we use the STL formula $\chi = \square_{[0, t_{max}]} \neg(|V_{DC}(t) - V_d| < \epsilon)$ to state that "the bus voltage V_{DC} should never deviate from the desired value V_d by more than a margin ϵ for more than t_{max}". Then, since V_{DC} should be in its range only after the initial start-up transient time τ_i, we require that $\phi(\tau_i) = \neg(\diamondsuit_{[\tau_i, \infty)} \chi)$ hold.

B. Control Design

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 [10]. The resulting controller has 113 states and was synthesized in approximately 2 s on an Intel Core i7 2.3-GHz processor. To analyze and optimize the real-time performance of the controller, we used a hybrid model implemented in Simulink, as shown in Fig. 2 (b), including the synthesized controller, imported as a Matlab function, and the other EPS components based on blocks from the SimPowerSystems library. Contactors were implemented to respond with a fixed delay T_d to the open/close commands from the BPCU. Moreover, we used a parametric version $\phi(\tau_i, \tau_e)$ of the formula $\phi(\tau_i)$ above, where t_{max} is replaced by τ_e, 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, we cast an optimization problem following the formulation in (1)

$$\min_{T_r>0} \quad 1/T_r \tag{2}$$
$$\text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{u}, V_{DC}, T_r) = 0 \\ V_{DC} \models \phi\left(\tau_i(T_r, T_d^*), \tau_e\right) \quad \forall \tau_e \geq t_{max} \quad \forall \boldsymbol{u} \text{ s.t. } \boldsymbol{u} \models \varphi_e' \end{cases}$$

where $\boldsymbol{C} = 1/T_r$ is the cost function, $\boldsymbol{\kappa} = T_r$ is the design parameter, $\phi_s(\boldsymbol{\pi}) = \bigwedge_{\tau_e \geq t_{max}} \{\phi\left(\tau_i(T_r, T_d^*), \tau_e\right)\}$ is the conjunction of PSTL formulas that must be satisfied, each parameterized by $\boldsymbol{\pi} = T_r$, and φ_e' refines the environment assumption formula φ_e of the LTL contract in Section III-A. In this case, the system behavior is the trace $\boldsymbol{s} = (\boldsymbol{u}, V_{DC})$, where V_{DC} is the output signal to be observed during simulation and $\boldsymbol{u}$ spans the set of all admissible failure injection traces that are consistent with the assumptions in Section III-A. The initial start-up transient time τ_i can be estimated from simulation as a function of T_r and T_d^*.

The formulation in (2) includes an infinite set of formulas that must be satisfied for all possible input traces and values of $\tau_e \geq t_{max}$. However, such formulation can be further simplified, by observing that (2) is equivalent to

$$\max_{T_r>0} \quad T_r \tag{3}$$
$$\text{s.t.} \begin{cases} \mathcal{F}(\boldsymbol{u}, V_{DC}, T_r) = 0 \\ \tau_e^*(T_r, T_d^*) \leq t_{max} \qquad \forall \boldsymbol{u} \text{ s.t. } \boldsymbol{u} \models \varphi_e' \end{cases}$$

where $\tau_e^*(T_r, T_d^*)$ is the maximum amount of time elapsed while the DC bus voltage is out of range, i.e. for how long at most the voltage requirement on the DC bus is violated. Such a maximum violation period can be computed as the

$$\sup\{\tau_e \geq 0 \mid \phi\left(\tau_i(T_r, T_d), \tau_e\right) = False\}. \tag{4}$$

As a further simplification, it is enough to compute $V_{DC}(t)$ and τ_e^* under the worst case input scenario, rather than for all possible input traces, whenever the worst case assumptions on $\boldsymbol{u}(t)$ can be determined a priori. Problem (3) can then be solved by first solving the optimization problem in (4) to compute τ_e^* as a function of T_r and T_d^* in the worst case input scenario, and then by computing the value T_r^* of the controller reaction time that makes τ_e^* equal to t_{max}.

Figure 3 shows the simulated voltage V_{B_3} of bus B_3 in Fig. 2 (a) as a function of time, for $T_r = 15$ ms, $T_d = 15$ ms and in the worst case scenario of cascaded faults in generators G_1, G_2 and rectifier R_1 (an event with probability $4 \cdot 10^{-17}$ based on our assumptions). The waveforms at the top and bottom of the figure are the voltage signals at the B_1 (AC) and B_3 (DC) buses, respectively. The signal in the middle represents the health status of R_1. Both the AC and DC voltages decay to zero because of the generators' faults. When a fault is also injected into R_1, an additional drop in the DC voltage is observed. The red signal at the bottom of the figure is interpreted as a Boolean signal, which is high (one) when χ holds (i.e. the requirement on the DC bus is violated) and low (zero) otherwise. The red signal represents the satisfaction of χ when $V_d = 28$ V, $\epsilon = 2$ V and $t_{max} = 70$ ms. The requirement on the DC bus is violated for 32 ms. Therefore, ($T_r = 15$ ms, $T_d = 15$ ms) is an unsafe parameter set.

The T_r versus T_d design space is explored in Fig. 4 (a) and (b) by sampling the parameter space in approximately 4 hours to populate a 13×13 point grid. The left plot represents the amount of elapsed time τ_e^* while the DC bus voltage is out of range, i.e. for how long the requirement on the DC bus is violated, as computed in (4). Such a violation period is then compared with the hard threshold $t_{max} = 70$ ms in the right plot, thus providing the designer with a "safe" region (marked in blue in Fig. 4) for selecting the controller clock as a function of the contactor delay. As an example, for $T_d = 20$ ms the maximum BPCU reaction time T_r^* allowed for safe operation is 4 ms.

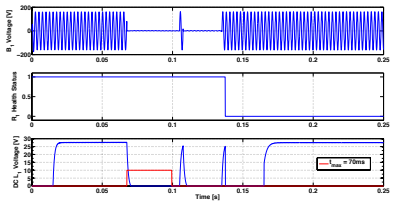
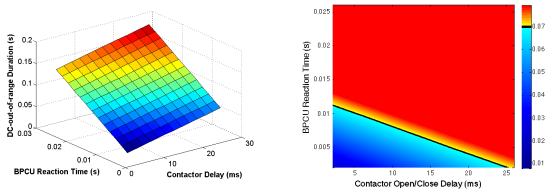


Fig. 3. Real-time requirement violation at the DC bus B_3 due to the failure of two generators and one rectifier in cascade.



(a) (b)

Fig. 4. BPCU reaction times and contactor delays in the blue region satisfy the DC bus requirement.

IV. Conclusions

The proposed methodology brings several innovations to current design practices, including requirement formalization with contracts, and coherent combination of provably-safe control synthesis techniques with design exploration and optimization steps. Future extensions of this work include developing tools to effectively guide designers towards requirement formalization as well as exploring techniques for cost-driven distributed synthesis of control protocols, to improve on both the optimality and the scalability of reactive synthesis approaches. Finally, incorporating control synthesis algorithms that can support richer specification languages (e.g., capturing transients, sensing and actuation delays) will also be considered as a future research direction.

References

[1] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract–based design for cyber–physical systems," in *Conf. Decision and Control*, Dec. 2011.

[2] K. Forsberg and H. Mooz, "System engineering for faster, cheaper, better," in *Center of Systems Management*, 1998.

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

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

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

[6] A. Pnueli, "The temporal logic of programs," in *Symp. Foundations of Computer Science*, vol. 31, no. 2, Nov. 1977, pp. 46–57.

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

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Massachussetts, USA: The MIT Press, 2008.

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

[10] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, and R. M. Murray, "TuLiP: a software toolbox for receding horizon temporal logic planning," in *Proc. Int. Conf. Hybrid Syst.: Computation and Control*, 2011.

[11] 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.

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

[13] H. Xu, U. Topcu, and R. M. Murray, "A case study on reactive protocols for aircraft electric power distribution," in *Int. Conf. Decision and Control*, 2012.