CHASE: Contract-Based Requirement Engineering for Cyber-Physical System Design

Pierluigi Nuzzo[1], Michele Lora[2], Yishai A. Feldman[3], Alberto L. Sangiovanni-Vincentelli[4]

[1] Department of Electrical Engineering, University of Southern California, Los Angeles. Email: nuzzo@usc.edu
[2] Department of Computer Science, University of Verona, Italy. Email: michele.lora@univr.it
[3] IBM Research, Haifa, Israel. Email: yishai@il.ibm.com
[4] Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. Email: alberto@berkeley.edu

***Abstract*—This paper presents CHASE, a framework for requirement capture, formalization, and validation for cyber-physical systems. CHASE combines a practical *front-end formal specification language based on patterns* with a rigorous *verification back-end based on assume-guarantee contracts*. The front-end language can express temporal properties of networks using a declarative style, and supports automatic translation from natural-language constructs to low-level mathematical languages. The verification back-end leverages the mathematical formalism of contracts to reason about system requirements and determine inconsistencies and dependencies between them. CHASE features a modular and extensible software infrastructure that can support different domain-specific languages, modeling formalisms, and analysis tools. We illustrate its effectiveness on industrial design examples, including control of aircraft power distribution networks and arbitration of a mixed-criticality automotive bus.**

I. Introduction

Safety-critical and time-critical cyber-physical systems (CPSs) pose several design and verification challenges [1], [2]. The design space is often too large and heterogeneous to be efficiently explored with strong guarantees of correctness, dependability, and compliance with regulations. While virtual prototyping and model-based engineering tools are the *de facto* standard for system development, the *concept design phase* largely remains a manual process. Modern requirement-management tools are still predominantly centered on text-based languages, often not in the mother tongue of the engineer, which creates opportunities for ambiguities, redundancies, and potential conflicts [3]. Different design stages tend to use domain-specific languages and tools that are poorly inter-operable, which makes it hard to combine the results of different analysis or synthesis methods. Assessing system correctness is then left to lengthy simulations and prototype tests later in the design process, which may yield implementations that are inefficient and sometimes do not even satisfy the requirements.

Contract-based design has recently emerged as a paradigm to address the above challenges and provide formal support for the design of complex systems [1], [4]. Contracts are mathematical objects that model the interface between components and levels of abstraction in a design and establish the foundations for assume-guarantee (A/G) reasoning about composability and abstraction/refinement relationships. Contracts enable modular and hierarchical verification of global properties of a system, whose satisfaction can be efficiently proven based on the satisfaction of local properties of the components [5]. Contracts support rigorous stepwise refinement, where hierarchical specifications can be used to reason about component decompositions when the component implementations are not yet available [6]. Contracts facilitate component reuse, as any components satisfying a contract directly inherit its guarantees. A few contract theories have been developed and demonstrated over the years [4], [6]. However, the development of tools that support contract-based design and enable its concrete adoption by system engineers is still in its infancy. This paper aims to bridge this gap by raising the level of abstraction of *design capture*. We introduce CHASE[1] (Contract-based Heterogeneous Analysis and System Exploration), a contract-based requirement engineering framework for system-level design exploration. CHASE combines a new front-end formal specification language based on patterns with a verification back-end based on contracts. The front-end language is used to capture requirements from natural-language constructs and translate them into contracts. The verification back-end allows assessing requirement correctness, completeness, and consistency by solving contract compatibility, consistency, and refinement checking problems. To the best of our knowledge, CHASE is the first contract-based framework addressing the end-to-end process from natural-language requirement capture to their formalization and validation. Our contributions can be articulated as follows:

- A framework that can *reason about temporal requirements on the behaviors of networks of dynamic components by leveraging assume-guarantee contracts* [4].
- A *new, declarative-style, formal language based on patterns*, i.e., predefined primitives from which mathematical specifications are automatically generated. While retaining a rigorous semantics, our language is more accessible to system engineers, and *amenable to translations from natural-language constructs*.
- Two *industrial case studies* that demonstrate the use of CHASE: the design of embedded controllers for aircraft electric power distribution networks [6] and the arbitration of a mixed-criticality automotive bus. We show that CHASE can substantially facilitate the *orchestration* of formal, exhaustive analyses of certain temporal properties of networks that would otherwise be lengthy, tedious, or error-prone to even formulate, let alone check.

Related Work. CHASE differs from previous end-to-end requirement engineering frameworks [7], [8], as it supports contracts as a specification theory, as well as a richer set of requirement constructs or analysis methods. A framework using modal interfaces, an automata-based formalism related to contracts, was also proposed [9]. Our approach is different since it focuses on a declarative-style formalism. A declarative style is often deemed better for high-level requirement validation, as it retains the correspondence between changes

This work was partially supported by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

[1]CHASE is available open-source at https://chase-cps.github.io/chase.

978-3-9819263-0-9/DATE18/©2018 EDAA

in the informal statements and their effect on the formal constructs. A declarative language, HRELTL (Hybrid Linear-time Temporal Logic with Regular Expressions), was proposed for hybrid system requirement validation based on contract refinement checking [5], [10]. CHASE instead proposes a unifying interface and infrastructure for the development of multiple contract-based reasoning tasks, including conflict detection and the feasibility of downstream artifacts satisfying the requirements, via the coordination of multiple decision procedures. Moreover, this paper demonstrates an end-to-end solution which raises the level of abstraction of requirement capture. Finally, with respect to previous pattern-based approaches to requirement capture (see, e.g., [11]–[14]), CHASE aims to introduce a higher-level language supporting behaviors of generic networks of dynamic components, which is closer to the abstractions used in system engineering [3].

II. PRELIMINARIES

A/G Contracts. We summarize the main concepts of the A/G contract framework [4] by starting with a generic representation of a component, i.e., an element of a design, characterized by a set of (input, output, or internal) *variables* and a set of *behaviors* over them. Components can be connected under constraints on the values of certain variables. A *contract* $\mathcal{C}$ for a component M is a triple (V, A, G), where V is the set of component variables, and A and G are sets of behaviors over V. 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 all the behaviors of M are *contained* in the guarantees of $\mathcal{C}$ once they are composed with the assumptions. We say that M is a (legal) *implementation* of $\mathcal{C}$. However, a component E can also be associated with a contract $\mathcal{C}$ as an *environment*. We say that E is a (legal) environment of $\mathcal{C}$ whenever all the behaviors of E are contained in the assumptions of $\mathcal{C}$. A contract is *consistent* when the set of implementations satisfying it is not empty; it is *compatible* if there exists a legal environment E for it.

To reason about replaceability between different abstraction layers in a design, contracts can be ordered via a *refinement* relation. We say that $\mathcal{C}$ refines $\mathcal{C}'$ if and only if $\mathcal{C}$ has weaker assumptions and stronger guarantees. We can then replace $\mathcal{C}'$ with $\mathcal{C}$. Contracts can be combined according to different rules. *Composition* of contracts can be used to construct complex global contracts out of simpler local ones. Reasoning on the compatibility and consistency of a composite contract can then be used to assess whether there exist components such that their composition is valid, even if the full implementation of the components is not available. Finally, the *conjunction* of contracts can also be defined to combine multiple requirements on the same component that need to be satisfied simultaneously. We refer the reader to the literature [4] for the formal definitions and mathematical expressions for the operations and relations summarized above.

Temporal Logic Contracts. The wealth of results in temporal logic and model checking can provide a substantial basis for requirement analysis for *discrete-time (discrete-event) discrete-state* system abstractions [15]. Both assumptions A and guarantees G of a contract $\mathcal{C}$ can be specified as temporal logic formulas [6]. 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* [15]. Composition and conjunction of contracts $\mathcal{C}_1$ and $\mathcal{C}_2$ can be represented by appropriate Boolean combinations of the formulas A_1, A_2, G_1, and G_2. *Refinement* is an instance of *validity checking*. Checking that $\mathcal{C}_1$ refines $\mathcal{C}_2$ can be translated into checking that $A_1 \rightarrow A_2$ and $G_2 \rightarrow G_1$ are valid formulas (i.e., tautologies for the language). Contract *compatibility* and *consistency* checking are, instead, less immediate, since they may either translate into checking *satisfiability* or *realizability* of formulas, depending on the specific temporal logic used and the semantics adopted for implementations and environments. As an example, for consistency checking of Linear Temporal Logic (LTL) contracts, we may be required to check whether there exists a system trace (run) that satisfies the contract or whether there exists a system (e.g., a state machine) for which all traces (runs) satisfy the contract. The former condition turns into testing whether $A \rightarrow G$ is satisfiable, while the latter requires testing whether $A \rightarrow G$ is realizable.

When contracts are expressed using LTL, algorithmic techniques from *reactive synthesis* [16], [17] can also be used to generate a discrete abstraction of the design from a consistent and compatible set of contracts. In this paper, we illustrate the capabilities of CHASE using a subset of LTL, namely Generalized Reactivity (1) (GR(1)) that generates synthesis problems whose run time is polynomial in the number of valuations of the contract variables [16]. Using LTL contracts, CHASE can directly reason about synchronous or asynchronous networks of components with discrete dynamics, thus capturing the computation and communication parts of a CPS. Moreover, it can support hierarchical approaches relying on abstractions [17] to reason about high-level requirements of hybrid dynamical systems. Finally, being centered on the abstract notion of behavior, the contract manipulation primitives in CHASE are flexible and can be customized to also support requirement analysis methods for richer logics, such as the ones recently proposed for HRELTL [10], STL (Signal Temporal Logic) [18], and StSTL (Stochastic Signal Temporal Logic) [19].

III. FORMAL SPECIFICATION LANGUAGE

We detail the components of the CHASE framework starting with the front-end formal specification language. The formal language of CHASE relies on a representation of the design as a network of elements and interconnections. Networks relate to different domains. For illustration purposes, we consider the aircraft power distribution network design problem and methodology [6], [20] which we also use as a case study in Sec. V. The primary distribution network of an aircraft Electric Power System (EPS) typically includes generators, AC buses, rectifiers, and DC buses, all connected through switches called *contactors*. A sample architecture is represented in Fig. 1. The generators power the buses and their loads (not shown in the figure). AC power is converted to DC power by rectifier units. The controller (not shown in the figure) monitors the availability of power sources and configures the contactors, such that essential buses remain powered even in the presence of failures. Given an EPS topology with fault sensors and a set of requirements, we are interested in the design of the controller, which runs a control strategy so that all the closed-loop system behaviors satisfy the requirements. Fig. 2 shows a description of the EPS architecture and system specification, expressed in CHASE's formal language. We use this example to discuss the main language constructs.

Component Specification. A *domain* is a set of component *types* (e.g., `generator`, `load`, `AC bus`) with arbitrary domain-specific names; each type is an instance of one of the *generic types* made available by the language (e.g., `Source`, `Sink`, `Switch`). Elements have also domain-specific *attributes* (e.g., `left`, `right`) that allow partitioning them into sets (groups) (e.g., `left load`). A network *architecture* is specified by providing a list of components, interconnections, and their descriptions.

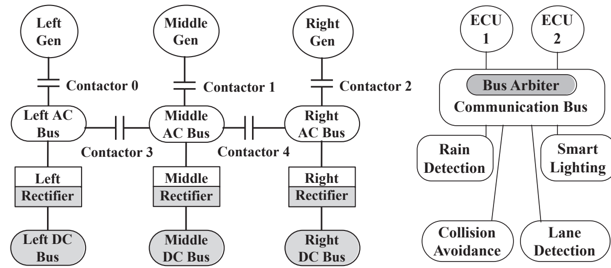

Fig. 1. Simplified electric power system (left) and automotive system (right) architecture examples used to illustrate the operation of CHASE.

```
1  TYPES
2  generator: Source(AC);
3  AC bus: Bus(AC);
4  DC bus: Bus(DC);
5  rectifier: Converter(AC, DC);
6  load: Sink(DC);
7  contactor: Switch;
8
9  COMPONENTS
10 left generator: 1;
11 left AC bus: 1;
12 left rectifier: 1;
13 contactor: *;
14 right generator: 1;
15 right AC bus: 1;
16 # etc.
17
18 ABBREVIATIONS
19 left generator: LG;
20 right generator: RG;
21 # etc.
22
23 CONNECTIONS
24 LG -(contactor 1)- LB;
25
26 SWITCHED(contactor)
27 RG -- RB;
28 AC bus -- AC bus;
29
30 UNSWITCHED
31 LB -- LR;
32 LR -- LD;
33 # etc.
34
35 REQUIREMENTS
36 must-disconnect-failed(generator);
37 # etc.
38
39 ASSUMPTIONS
40 max-failures(generator, 1);
41 # etc.
```

Fig. 2. Problem specification example using the CHASE formal language.

The CHASE generic types have special semantics, which may be used by both the reasoning modules (back-end) and the input module (front-end) for checking input consistency. The annotations in parentheses are optional. For instance, `Source(X)` is a source of the network and is characterized by only one connection, treated as an output. If the optional annotation `X` is supplied, it must be present on the connected component. For example, an `AC bus` can be connected to a generator of type `Source(AC)`, but a `DC bus` may not. Symmetrically, `Sink(X)` is a sink of the network and its single connection is treated as an input. `Bus(X)` is a bus that can be connected to multiple components. However, a bus of type `Bus(AC)` can be connected to multiple loads of type `Sink(AC)`, but not to loads of type `Sink(DC)`. `Switch(X)` is a component that can dynamically close or open a connection in the network, and it must be connected to exactly two other components. If the optional annotation `X` is supplied, it must be present on both connected components. `Converter(X, Y)` is a component that can have exactly two connections of different types. Finally, a generic type `Component(X)` is also provided, which can have exactly two connections of type `X`.

Individual components have names composed of one or more words, with one optional number. The rightmost word is the type while other words and the number are attributes. For example, `left AC bus` is a `bus` with attributes `left` and `AC`; `contactor 1` is a `contactor` with attribute `1`. If two words are used as a single concept (e.g., a type or an annotation) they are hyphenated. For instance, it is possible to instantiate an `auxiliary-power generator`, which is a `generator` with a single attribute `auxiliary-power`. By convention, all components' names are singular, not plural, even if they may be used to refer to sets of components in other sections of the problem specification (e.g., `CONNECTIONS`, `UNSWITCHED`, `REQUIREMENTS`). Components are listed with their full names, excluding numbers, and the number of components under that name. For instance, it is possible to declare that three left rectifiers are part of the network by listing `left rectifier: 3`. Each individual component can then be named by adding a number between 1 and the given number, e.g., `left rectifier 1`, `left rectifier 2`, `left rectifier 3`. The number will be missing in the name if only one component is specified, as in Fig. 2. An asterisk is used to denote a number that is unknown or it is not required to be bounded, as for the `contactor` component in Fig. 2; an arbitrary number, e.g., `contactor 100`, can then be used in the component name.

Architecture Specification. The system architecture, i.e., interconnection of components, is specified using the `CONNECTIONS`, `SWITCHED`, and `UNSWITCHED` sections. Components in these sections can be specified using their types, with any number of attributes. The connection specification refers to all the components having the given attributes, except that no self-connections are allowed. In the example of Fig. 2, `AC bus -- AC bus` means that any two different AC buses are connected. Connections can be switched or unswitched. In the former case, it is necessary to specify the type of switch or give an instance of that type. An instance can be given in parentheses between dashes, e.g., `LG -(contactor 1)- LB`. It is also possible to use a default name for the switch instances and specify the connection using two dashes, e.g., in `RG -- RB`. All connection specifications following a declaration `SWITCHED(type)` will use switches of the given type, with automatically-generated names. All connection specifications following a declaration `UNSWITCHED` will not contain switches.

Requirements Patterns. There can be many patterns for a domain, all expressed as predicates relating to various components. An excerpt of requirements and assumptions of the EPS example appears in Fig. 3. Patterns refer to sets of components (e.g., `AC bus`) rather than types. For instance, `never-connect(X, Y, Z)` predicates that there shall not be a closed connection between an element of `X` and an element of `Y` that goes through an element of `Z` (which is optional). The pattern `must-disconnect-failed(X)` predicates that elements of set `X` must be disconnected when not healthy. This assumes that a switch is available on every edge connected to an element of `X`. If this assumption is relaxed, additional parameters should be added. The pattern `prefer-active-connection(X, Y)` states that, if possible, each element of `X` should have at least one *active path* to at least one element of `Y`, meaning that all the switches should be closed along the path and all components, including switches, should be healthy (operational). The pattern `always-active-connection(X, Y)` requires that every element of `X` have at least one active path to at least one element of `Y`. Requirements will be internally translated into contract assumptions and guarantees. Several assumptions, e.g., the ones that are inherently linked to the network topology, are automatically added as a part of the language interpretation and translation process. It is, indeed, impractical to directly require the user to explicitly determine these "hidden" assumptions which are implicitly taken during the specification process. However, it is possible to define additional `ASSUMPTIONS`, e.g., to force a desired scenario or specify legal environments. For instance, `no-recovery(X)` states that if an element of `X` becomes unhealthy, it never becomes healthy again, while `max-failures(X, N)` says that no more than `N` elements of `X` can fail simultaneously, `N` being a non-negative integer.

Natural-Language Requirements. CHASE supports the translation of natural-language requirements and assumptions into the formal language presented above in order to make contract-based reasoning more accessible to requirements engineers. The natural-language statements from which the specification of Fig. 3 was generated appear in Fig. 4. We use the English Slot Grammar (ESG) parser [21] with a set of parse-tree patterns to convert the natural-language requirements into a semantic representation, from which we generate the spec-

```
 1  REQUIREMENTS
 2  prefer-active-connection(left DC bus, left generator);
 3  must-disconnect-failed(generator, 20, MS);
 4  never-connect(generator, generator, AC bus);
 5  always-active-connection(DC bus, generator, 30, MS);
 6
 7  ASSUMPTIONS
 8  no-recovery(generator);
 9  no-failures(AC bus);
10  no-failures(Rectifier);
11  no-failures(DC bus);
12  no-failures(load);
13  switch-on-time(contactor, 10, MS);
14  switch-off-time(contactor, 10, MS);
15  max-failures(generator, 1);
```

Fig. 3. Excerpt of the requirements for the aircraft power system example, translated to the CHASE formal language.

```
 1  REQUIREMENTS
 2  If possible, left DC buses shall be powered by left generators.
 3  If the left generator fails, the left AC bus shall be connected to another generator.
 4  Failed generators must be disconnected in 20 ms or less.
 5  Generators shall never be connected in parallel through AC buses.
 6  When a contactor receives an open signal, it shall become open in 10 ms or less.
 7  When a contactor receives a close signal, it shall become closed in 10 ms or less.
 8  A DC bus shall never be disconnected from a generator for more than 30 ms.
 9
10  ASSUMPTIONS
11  Generators do not recover from failures.
12  AC buses do not fail.
13  Rectifiers do not fail.
14  DC buses do not fail.
15  Loads do not fail.
16  At most 1 generator may fail.
```

Fig. 4. Excerpt of the natural-language requirements for the power system.

ifications in the formal language summarized in this section. The same natural-language analysis technology was previously used in a study about the use of cognitive technologies in requirements analysis [3], where more details can be found. While we leverage natural-language analysis as a design aid to enable automatic generation of formal specifications, requirement formalization does not exclusively rely on this technology. The translation engine cannot cover all ways in which the requirements can be expressed, and it is possible that the tool's understanding will be different from the engineer's intent. If there are ambiguities, the designer can change the text until the interpretation is correct. Alternatively, he or she can choose to bypass the natural-language translation and manually specify the requirements in the CHASE formal language.

IV. Architecture and Verification Back-End

A representation of the main CHASE components is shown in Fig. 5. We provide details on the modular infrastructure and some mechanisms used to reason about temporal properties of networked discrete systems.

Software Architecture. The software infrastructure of CHASE is based on three main components: a set of front-end modules, the CHASE libraries, and a set of back-end modules. The *front-end modules* allow parsing different specification formats and domain-specific languages and build the CHASE internal representation of a design problem. Current support for the formal language described in Sec. III is based on the ANTRL4 [22] parser generator. The *CHASE libraries* provide a set of classes, data structures, and methods to represent and manipulate system design problems in a contract framework. The *Architecture* library allows specifying cyber-physical system architectures and network topologies according to a graph-based formalism previously introduced in the literature [6]. The *Behaviors* library provides constructs to represent the behaviors of the system components: their variables, ports, variable and port types, dynamics, and temporal properties. For example, this library enables the specification of propositional logic, first-order logic, and LTL formulas. The *Contracts* library contains data structures to model design and verification problems as compositions, conjunctions, and refinements of contracts. The *Specification* library provides mechanisms to specify the requirement validation problem, including libraries

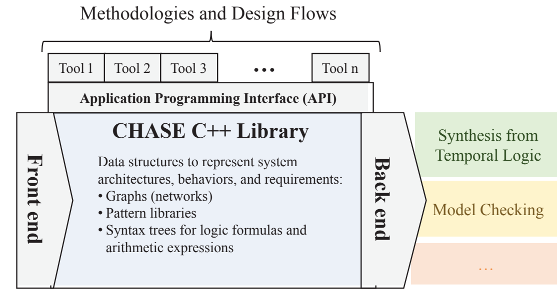

Fig. 5. The CHASE framework components and flow.

of standard requirements and patterns. The *Manipulation* library provides a set of methods for robust manipulation of the classes and data structures used in the other libraries, e.g., methods to manipulate and analyze graphs, process timing intervals in the specifications, visit the internal representation of the system. Tool developers as well as system designers can leverage the Application Programming Interface (API) provided by the CHASE libraries to build customized tools and methodologies for specific problems. Finally, a set of *back-end modules* allows translating the CHASE representation into appropriate languages supported by formal verification and synthesis routines that are used to solve contract compatibility, consistency, and refinement checking problems.

Reasoning about Temporal Properties. We illustrate the capabilities of CHASE by referring to an internal requirement representation based on LTL A/G contracts. CHASE can formulate LTL satisfiability, validity, and synthesis problems from requirements expressed in its formal language. We use the TuLiP [17] toolbox to solve realizability problems and provide an implementation of the design in the form of a Python module, when the requirements are consistent. While LTL does not directly support reasoning about time intervals, CHASE supports timing specifications using *counters*. A counter is an integer variable with fixed span that, at each instant, can only be incremented by one unit or reset to zero. Given a set of requirements including real time intervals, it is then possible to select a discretization step τ as the greatest common divisor (GCD) of all the lengths of the time intervals used in the specification. The span N of a counter associated with an interval (e.g., representing a delay) is then determined by the ratio between the length T of the interval and the discretization step, i.e., $N = T/\tau$. All LTL formulas are then interpreted over sequences of valuations (assignments) over system variables, where the time interval between two consecutive valuations has length τ. CHASE distinguishes between *physical counters*, used to model physical properties of the system, such as actuation or sensing delays, and counters that are used to measure the time elapsed between the occurrence of two events, called *monitor counters*. Physical counters affect the contract assumptions, and they allow expressing assumptions that may depend on both input and output variables of components. For example, a physical counter D modeling the delay between a control action C and its expected change in state variable S augments the contract assumptions with the conjunction of the following LTL formulas, extended with arithmetic predicates over integers:

$$A \leftrightarrow (S \wedge C), \quad B \leftrightarrow (\neg S \wedge C), \quad \Box(A \rightarrow \bigcirc(D = 0)) \tag{1}$$

$$\bigwedge_{n=0}^{d-2} \Box((B \wedge D = n) \rightarrow \bigcirc((B \wedge D = n+1) \vee (A \wedge D = 0))) \tag{2}$$

$$\Box((B \wedge D = d-1) \rightarrow \bigcirc(A \wedge D = 0)) \tag{3}$$

where A, B, S, and C are Boolean variables, and $\Box$ and $\bigcirc$ correspond to the LTL temporal operators *always* and *next*. The integer constant d is the maximum delay assumed, i.e., the value of D ranges from 0 to $d-1$. Overall, these formulas express that S will be true whenever C is true after a delay smaller than or equal to d. A similar conjunction of formulas is used to express the delay after which S will be false whenever C is false. Finally, a similar procedure can be used to model monitor counters, appearing in the contract guarantees, e.g., to model the fact that the state S must be true within an amount of time t after C becomes true. Counters allow extending the abstraction provided by LTL to also capture a model of the dynamics of the components or the real-time constraints on their operation. They are automatically instantiated by CHASE in a way that is transparent to the user.

V. Application Examples

Aircraft Power Distribution System. In an aircraft EPS, a supervisory controller must actuate a set of contactors to distribute power from generators to loads and satisfy the demand for a set of predetermined flight conditions and faults [6]. We require that essential loads and buses (such as flight-critical actuators) never be unpowered for more than a specified time t_{max}. Validation of the closed-loop system requirements to determine the feasibility of a control algorithm cannot be performed correctly unless we account for the sensing and actuation delays in both the physical plant and the embedded platform as well as the possible failure scenarios. This is made possible by CHASE.

Fig. 1 illustrates a simplified version of the system including three power sources (left, right, and middle generators), three AC buses, three rectifiers, and three DC buses, all connected through contactors. An excerpt of the natural-language requirements examined in this case study is provided in Fig. 4, while Fig. 3 shows an excerpt of their translation into the CHASE language. To avoid generator damage, we proscribe AC sources to be paralleled, i.e., no AC bus can be powered by multiple generators at the same time (`never-connect` pattern). A bus connected to an unhealthy source may cause a short-circuit failure; therefore, we require that appropriate contactors open when a generator becomes unhealthy to isolate it and prevent its use (`must-disconnect-failed` pattern). We then test the consistency of a contract for the EPS controller that can accommodate a failure in one generator, by rerouting power from another generator to the corresponding DC bus in a time interval which is less than or equal to $t_{max} = 30$ ms (`always-active-connection`). Both the plant topology and the controller should be designed to be robust to certain combinations of faults potentially causing the failure of an essential component. To this effect, in the `ASSUMPTIONS` section, we can explore different scenarios by enumerating the maximum number of components that are allowed to fail for each type of component (`max-failures` and `no-failures`). These assumptions directly translate into environment configurations that may occur and must be counteracted by the controller in due time. Finally, we assume that when a component fails during the flight, it will not come back online (`no-recovery`).

To capture the physical dynamics of the plant and the real-time constraints on the system operation, each contactor requires two additional counter variables to model opening and closing delays. Similarly, each generator requires one counter variable to model the delay with which it is disconnected, while one counter variable is needed per each essential bus, to capture the requirement on the maximum time t_{max} allowed for a bus to stay unpowered. These counter variables are introduced by CHASE in a way that is transparent to

TABLE I. Synthesis time (s) as the number of counters (vertical axis) and their span (horizontal axis) increase.

	3	4	6	8	10	15	30
6	0.03	0.01	0.05	0.1	0.1	0.21	0.37
8	0.03	0.02	0.1	0.11	0.27	1.05	18.94
10	0.06	0.03	0.16	0.24	2.58	64.52	27383.95
12	0.07	0.03	0.42	0.66	23.3	5694.8	> 24 h
14	0.12	0.11	1.75	2.83	439.71	> 24 h	> 24 h
16	0.1	0.09	10.04	20.15	19775.75	> 24 h	> 24 h

the user, when processing the parameters provided with the `must-disconnect-failed`, `switch-on-time`, and `switch-off-time` patterns. CHASE finds that the first six requirements in Fig. 4 are always consistent, while the addition of the seventh requirement (line 8) may create inconsistencies when the time parameter t_{max} (set to 30 ms in Fig. 4) is less than 20 ms. By simulation of a synthesized controller, it is possible to infer some of the scenarios that expose this inconsistency. For example, let us assume that all the generators are serviceable, and each AC bus receives power from the corresponding source (line 2 in Fig. 4) until the left AC generator becomes unserviceable. Contactors 0 and 1 are not actuated at the same time to avoid connecting two power sources (line 5); it is then possible that a delay of 20 ms is reached until contactor 0 is open and contactor 1 is closed (lines 3, 4, 6, and 7). This will cause a violation of the requirement in line 8 for $t_{max} < 20$ ms.

Tab. I shows the synthesis time, i.e., the time required to create a contract implementation, as a function of the number of counters in the system (vertical axis) and their span N (horizontal axis). The number of counters is incremented with a step of 2 to model (opening or closing) delays for the five contactors in Fig. 1. In all cases, we assume that, if a contactor has no counter associated with it, it behaves as an "ideal" switch and presents minimum delay, i.e., can be closed or opened within the discretization step τ. For the experiments in Tab. I, it was enough to change the timing parameters of the same, compact specification template in Fig. 3. This should be contrasted with the lengths of the generated LTL formulas, ranging from 540 to 4,670 literals, a challenging size to handle manually. While the synthesis time grows exponentially with both the number of counters and their span, checking realizability, without generating an implementation, required much less time, from 0.05 to 1.8 s on a 3.4-GHz Intel Core i7 processor with 16-GB RAM, since it can be performed by adopting symbolic algorithms to reason about sets of states without enumerating all of them.

Automotive Communication System. Modern automotive systems have evolved to cyber-physical systems, consisting of heterogeneous electronic control units (ECUs), sensors, and actuators, that communicate over a network of buses. Various functions are realized by distributed tasks which communicate by exchanging messages over the shared buses, leading to mixed-criticality systems where multiple functions with different criticality levels can be supported by one ECU and one function can be distributed over multiple ECUs [23].

We apply CHASE to a requirement validation problem for an automotive communication system that provides assistance to the driver by monitoring a set of functions located in the front of a car. As shown in Fig. 1, the system consists of four subsystems: a rain sensing system for automatic windscreen wipers management, a collision avoidance system, a lane departure warning system, and a smart light management system. The sensors of each subsystem send data to two ECUs. Data may be sent in the form of high-priority or low-priority signals (messages) and are propagated to the ECUs via a shared double-priority bus. We require that high-priority signals are processed within 5 ms, while low-priority requests

```
 1 TYPES
 2 ECU: Sink(data);
 3 communication bus: Bus(data);
 4 peripheral: Bus(data);
 5 low iface: Converter(low, data);
 6 high iface: Converter(high, data);
 7 high signal: Source(high);
 8 low signal: Source(low);
 9 nw switch: Switch;
10 cpu switch: Switch;
11
12 COMPONENTS
13 main ECU: 2;
14 main data bus: 1;
15 sensor peripheral: 4;
16 rain low iface: 1;
17 rain low signal: 1;
18 # etc.
19
20 ABBREVIATIONS
21 main ECU 1: ECU1;
22 main data bus 1: DBUS;
23 sensor peripheral 1: RAIN;
24 rain low iface 1: RLI;
25 # etc.
26
27 CONNECTIONS
28 DBUS -(cpu switch 1)- ECU1;
29 RAIN -(nw switch 1)- DBUS;
30 RLI -(nw switch 5)- RAIN;
31 # etc.
32
33 REQUIREMENTS
34 never-connect(ECU, ECU, communication bus);
35 never-connect(peripheral, peripheral, communication bus);
36 never-connect(high iface, low iface, peripheral);
37 always-active-connection(high signal, ECU, 5, MS);
38 always-active-connection(rain low signal, ECU, 14, MS);
39 always-active-connection(collision low signal, ECU, 12, MS);
40 always-active-connection(lane low signal, ECU, 12, MS);
41 always-active-connection(light low signal, ECU, 14, MS);
42
43 ASSUMPTIONS
44 initial-state(nw switch 1, 1);
45 initial-state(nw switch 6, 1);
46 initial-state(cpu switch 1, 1);
47 no-failures(*);
48 switch-on-time(Switch, 1, MS);
49 switch-off-time(Switch, 1, MS);
50 # etc.
```

Fig. 6. Excerpt of the requirements for the automotive system example.

may be accommodated at a slower pace. The ECUs are able to guarantee a worst-case processing delay of 1 ms, i.e., any request coming from the bus will be serviced within 1 ms. We want to check whether the requirements are consistent and whether there exists an arbitration strategy for the Bus Arbiter.

Fig. 6 shows an excerpt of the formal specification in CHASE's language. New component types define the new application domain. For example, the ECUs are instances of type `Sink` and must be connected to components that send data, i.e., that are labelled with the `data` annotation. The main Communication Bus and the four peripherals are instances of type `Bus` and may also be only connected with sources or sinks annotated with `data`. Two types of interfaces (`high iface` and `low iface`) are used to convert high-priority and low-priority signals of types `high` and `low`, respectively, generated from a `high` and `low signal` source, into signals of type `data`, accepted by a `peripheral` bus. Elements of type `Switch` are used by the Arbiter to regulate the network traffic by allowing or disallowing signal or data transfers between certain pairs of system components. To assure that high-priority messages are correctly transferred, there should be a link between each ECU and each peripheral at least every 5 ms (`always-active-connection` pattern in line 37), while low-priority messages are subject to more relaxed deadlines (lines 38–41).

After developing the first case study in this section, it took approximately 30 minutes to generate and validate the specification in Fig. 6. CHASE generated and performed a consistency check for a contract consisting of an LTL formula with about $3,000$ literals in 22 s. The specification was found to be inconsistent, and a possible way to resolve the conflict was to increase the tolerance for the low-priority signals from the lane departure detection system from 10 to 12 ms. In this case, an arbitration algorithm was generated in about 1 minute and the resulting finite state machine has 199 states and 396 transitions.

Overall, the design examples discussed in this section show that both natural-language translation and pattern-based specification language are useful to hide the details of the requirement formalization, which can be massive, and therefore reduce the chances of errors. CHASE's formal language can be reused across domains of applications, which is an indication of its power to capture the essence of several problems of interest in CPS design. Finally, the CHASE platform can make it possible to perform extensive evaluation of different requirement validation approaches.

VI. CONCLUSIONS

We introduced CHASE, a framework and approach to enable validation of high-level system specifications including complex behaviors. Future work includes extensions to support more formalisms, and developing mechanisms to convey interpretable explanations for conflicting requirements.

REFERENCES

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

[2] J. Sifakis, "Rigorous system design," *Foundations and Trends in Electronic Design Automation*, vol. 6, no. 4, pp. 293–362, 2013. [Online]. Available: http://dx.doi.org/10.1561/1000000034

[3] Y. A. Feldman and H. Broodney, "A cognitive journey for requirements engineering," in *Ann. INCOSE Int. Symp.* INCOSE, Jul. 2016.

[4] A. Benveniste *et al.*, "Contracts for System Design," INRIA, Rapport de recherche RR-8147, Nov. 2012.

[5] A. Cimatti and S. Tonetta, "Contracts-refinement proof system for component-based embedded systems," *Science of Computer Programming*, vol. 97, Part 3, pp. 333 – 348, 2015.

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

[7] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner, "ARSENAL: automatic requirements specification extraction from natural language," in *NASA Formal Methods Symposium*, 2016, pp. 41–46.

[8] J. Badger, D. Throop, and C. Claunch, "VARED: verification and analysis of requirements and early designs," in *Requirements Engineering Conf.*, 2014, pp. 325–326.

[9] B. Caillaud, "Mica: A modal interface compositional analysis library," http://www.irisa.fr/s4/tools/mica, Oct. 2011.

[10] A. Cimatti, M. Roveri, and S. Tonetta, "Requirements validation for hybrid systems," in *Proc. Int. Conf. Computer Aided Verification*, 2009, vol. 5643, pp. 188–203.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proc. Int. Conf. Software Engineering*, 1999, pp. 411–420.

[12] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," 2011, pp. 1023–1028.

[13] K. C. Castillos, F. Dadeau, J. Julliand, B. Kanso, and S. Taha, "A compositional automata-based semantics for property patterns," in *Int. Conf. Integrated Formal Methods*, 2013, pp. 316–330.

[14] K. Y. Rozier, "Specification: The biggest bottleneck in formal methods and autonomy," in *Int. Conf. Verified Software: Theories, Tools, and Experiments*, 2016, pp. 8–26.

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

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

[17] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray, "Control design for hybrid systems with TuLiP: The temporal logic planning toolbox," in *IEEE Conf. Control Applications*, 2016.

[18] S. Ghosh *et al.*, "Diagnosis and repair for synthesis from signal temporal logic specifications," in *Proc. Int. Conf. Hybrid Systems: Computation and Control*, 2016.

[19] J. Li, P. Nuzzo, A. Sangiovanni-Vincentelli, Y. Xi, and D. Li, "Stochastic contracts for cyber-physical system design under probabilistic requirements," in *Int. Conf. Formal Methods and Models for Co-Design*, 2017.

[20] P. Nuzzo, A. Sangiovanni-Vincentelli, D. Bresolin, L. Geretti, and T. Villa, "A platform-based design methodology with contracts and related tools for the design of cyber-physical systems," *Proc. IEEE*, vol. 103, no. 11, Nov. 2015.

[21] M. C. McCord, J. W. Murdock, and B. K. Boguraev, "Deep parsing in Watson," *IBM J. Res. Dev.*, vol. 56, no. 3, pp. 264–278, May 2012.

[22] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[23] D. Goswami *et al.*, "Challenges in automotive cyber-physical systems design," in *Int. Conf. Embedded Computer Systems*, 2012.