Iannopollo, Antonio; Tripakis, Stavros; Sangiovanni-Vincentelli, Alberto

Constrained synthesis from component libraries

Published in:
Science of Computer Programming

DOI:
10.1016/j.scico.2018.10.003

Published: 15/02/2019

Document Version
Publisher's PDF, also known as Version of record

Published under the following license:
CC BY-NC-ND

Please cite the original version:
Iannopollo, A., Tripakis, S., & Sangiovanni-Vincentelli, A. (2019). Constrained synthesis from component libraries. *Science of Computer Programming*, *171*, 21-41. https://doi.org/10.1016/j.scico.2018.10.003

Constrained Synthesis from Component Libraries

Antonio Iannopollo[a,*], Stavros Tripakis[b,c], Alberto Sangiovanni-Vincentelli[a]

[a]EECS Department, University of California at Berkeley, Berkeley, CA, USA
[b]Department of Computer Science, Aalto University, Aalto, Finland
[c]CCIS, Northeastern University, Boston, MA, USA

Abstract

Synthesis from component libraries is the problem of building a network of components from a given library, such that the network realizes a given specification. This problem is undecidable in general. It becomes decidable if we impose a bound on the number of chosen components. However, the bounded problem remains computationally hard and brute-force approaches do not scale. In this paper, we study scalable methods for solving the problem of bounded synthesis from libraries, proposing a solution based on the Counterexample-Guided Inductive Synthesis paradigm. Although our synthesis algorithm does not assume a specific formalism *a priori*, we present a parallel implementation which instantiates components defined as Linear Temporal Logic-based Assume/Guarantee Contracts. We show the potential of our approach and evaluate our implementation by applying it to two industrial-relevant case studies.

Keywords: counterexample-guided inductive synthesis, assume-guarantee contracts, component libraries, linear temporal logic, platform-based design

1. Introduction

While synthesis of an implementation given formal specifications in areas such as program synthesis is a well-studied problem [1–7], the application of synthesis techniques for Cyber-Physical Systems (CPS), where it is hard to completely decouple cyber and physical aspects of design, is still in its infancy.

Synthesis from component libraries is the process of synthesizing a new component by composing elements chosen from a library. This type of synthesis is able to capture the complexity of CPS by restricting possible synthesis outcomes to a set of well-tested, already available components. Library-based design approaches are nowadays a *de facto* standard in many fields, such as

*Corresponding author
Email addresses: antonio@berkeley.edu (Antonio Iannopollo), stavros.tripakis@gmail.com (Stavros Tripakis), alberto@berkeley.edu (Alberto Sangiovanni-Vincentelli)

VLSI design, where the market for Intellectual Property (IP) blocks is growing well above 3 Billion US$ [8]. On the basis of this trend the interest of system companies on library-based design, both for hardware and software, is steadily increasing. This leads to the need for methodologies which guarantee *correct-by-construction* designs.

The general problem of synthesis from component libraries, where the components are state machines, is undecidable [2]. In this paper, we focus on a decidable variant of the problem, where an explicit bound on the number of components in a solution is provided. Our goal is to find a composition of components which satisfies a specification while minimizing a certain cost function.

Although no particular formalism is assumed *a priori*, we cast a concrete version of this problem using *Linear Temporal Logic* (LTL)-based *Assume/Guarantee* (*A/G*) *Contracts* as the underlying specification of components. We then show how it is possible to solve this problem presenting two variants of an algorithm, a sequential and a parallel one, based on the *Counterexample-Guided Inductive Synthesis* (CEGIS) paradigm [3, 7]. To reduce the solution search space, this algorithm leverages designer hints, types, and other constraints over components, possibly *precomputed* and stored in the libraries as additional composition rules. To the best of our knowledge, this is the first time that a concurrent synthesis algorithm is proposed for this problem, thanks to the decoupling of a solution *topology* from its semantic evaluation.

The implementation of the parallel version of the algorithm resulted in a tool called PYCO, able to exploit multiprocessor computer architectures to speed up synthesis. We evaluate PYCO by synthesizing two industrial-relevant designs: first the controller, including both architectural and software aspects, of a *Brushless DC Electric Motor* (BLDC), and then the controller software for an aircraft *Electrical Power distribution System* (EPS) [9]. This last problem, in particular, has already been studied using contracts [10, 11]. In these papers, however, contracts have been used mostly for verification and to describe requirements, without playing any role in the controller synthesis process itself, performed using standard reactive synthesis techniques. Here, contracts collected in the component library represent controllers for a number of EPS subsystems. Our synthesis algorithm, for the first time, operates directly on those contracts to compose a controller that satisfies all the requirements.

The contributions of this paper, both theoretical and methodological, can be then summarized as: (i) definition and analysis of the problem of constrained synthesis from component libraries (CSCL); (ii) design and implementation of an algorithm to solve the CSCL problem when components are defined as LTL A/G contracts, leveraging *precomputed*, library-specific composition rules; and (iii) its application to two industrial-relevant case studies, i.e., synthesis of a controller for a brushless DC electric motor and an aircraft EPS.

The rest of the paper is organized as follows. Section 2 provides references on background concepts and describes related works, while in Section 3 we provide some preliminaries on A/G contracts. In Section 4 we define the synthesis problem we tackle and analyze its complexity, introducing a running example to explain in detail the problem encoding and the approach we adopt. We propose a solution for a concrete version of the problem in Section 5 and discuss implementation aspects in Section 6, including the description of the parallel variant of our algorithm. In Section 7 we present the case studies and empirical results. We discuss how our work can fit in more general design methodologies in Section 8, and we draw conclusions in Section 9.

This paper is a revised and extended version of our previous work published in [12]. Additions in this version include:

- extension of our framework and problem formulation by introducing cost functions;
- loosening of some of the synthesis constraints to allow for more flexibility in the synthesis process, at the cost of a slightly wider search space;
- addition of a new case study and extension of previous results.

2. Previous work

Synthesis. Our work on synthesis from component libraries is inspired by two major contributions in this field:

- In [1], Pnueli and Rosner show that the problem of synthesizing a set of distributed finite-state controllers such that their network, which is given and fixed, satisfies a given specification is undecidable. In that work, each component in the network is controlled by a finite-state machine (or *program*), and the goal is to synthesize programs that cooperate to satisfy a certain linear temporal logic formula φ;
- In [2], Lustig and Vardi show that the problem of synthesis from component libraries for *data-flow* compositions is also undecidable. Here components are *transducers*, i.e., finite-state machines able to map a set of input strings to a set of output strings. The specification, also in this case, is a linear temporal logic formula φ. In data-flow compositions, the output of a component is fed to another one, while all the components work synchronously to satisfy the specification. The paper also analyzes another type of composition, *control-flow*, where each component takes full control of the system for a certain period of time, before releasing the system resources to the other components.

Thus, [1] shows that fixing the topology of the network while letting the synthesis process find the components is undecidable, while [2] shows that fixing the components while letting the synthesis process find the topology (possibly by replicating components) is also undecidable. In our work, we too focus on data-flow compositions. Our components are, however, more general as we don't necessarily require specific formalisms (as programs or transducers) to define components and the system specification, although in our implementation we choose to use LTL A/G contracts. Yet, the undecidability results in [1] and [2] are relevant for us as they define the theoretical frame of this work. In our paper, we achieve decidability by imposing a bound on the total number of component instances, which are chosen from a library, positioning our efforts in between the two approaches presented above.

The general idea of synthesis from component libraries adopted here is reminiscent of the work in [5, 6]. There, Jha et al. considered the problem of synthesis of finite loop-free programs from libraries of atomic program statements. Our work is different, however, as (i) we consider a more generic concept of components (e.g., they can be also defined using temporal logic), considering multiple output ports and port types, (ii) decouple topological properties from the formalism used to describe the component, and (iii) introduce the idea, in the context of synthesis from component libraries, of applying library-specific composition rules to the synthesis problem.

A different perspective in synthesis from component libraries has also been described by Alur *et al.* in [13]. There, a controller is built out of library components in a control-flow fashion (using the terminology introduced in [2] and discussed above). That approach, while being relevant in the broader topic of synthesis from component libraries, is orthogonal to ours since we focus on data-flow compositions.

Relevant is also the extensive work done in the area of Supervisory Control Synthesis (SCS) [14]. SCS is the problem of synthesizing a controller for a discrete event system, i.e., an automaton, which exposes some controllable and uncontrollable behaviors. The specification defines which behaviors are admissible, and the goal of the controller is to restrict the controllable behaviors of the discrete event system in a way that satisfies the specification. Existing SCS algorithms, however, do not deal with libraries of components but, instead, their goal is to synthesize a controller *ex novo*. Here, we provide a more generic notion of components and focus our effort in synthesizing a controller through the composition of existing components.

In [15], Ramesh *et al.* focus on the problem of synthesis of embedded designs from component libraries. In that work, components are represented exclusively by their interface and connections are made on the basis of static relations between component ports. Given a specification, a particularly rich type system takes care of efficiently pruning the search space by solving a constraint satisfaction problem. Although our type system is not as expressive, our approach is more general as we consider components described by more complex specifications (not necessarily static) in addition to their interface.

Counterexample-Guided Inductive Synthesis, and the combination of Induction, Deduction, and Structure. CEGIS is a well-known synthesis paradigm which originates from techniques of debugging using counterexamples [16] and *Counterexample-Guided Abstraction Refinement* (CEGAR) [17]. CEGIS is an inductive synthesis approach where synthesis is the result of inferring details of the specification from I/O examples, driven by counterexamples usually provided by a constraint solver. In CEGIS an iterative algorithm, according to a certain concept class, generates candidate solutions which are processed by an oracle and either declared valid, in which case the algorithm terminates, or used as counterexamples to restrict the search space. CEGIS has been successfully used in a number of research areas, including program synthesis and sketching [3, 5, 6], and synthesis and completion of distributed protocols [18–20].

Recently, a novel methodology which formalizes the combination of *Structure, Inductive and Deductive* reasoning (SID) has been proposed in [21], representing a generalization of both CEGAR and CEGIS. The approach proposed in this paper instantiates the CEGIS paradigm (not a trivial task, in general), and thus it is an implementation of the SID methodology.

Platform-Based Design. Platform-Based Design (PBD) [22] is an iterative design methodology which has been successfully applied in a number of domains, including electronic and automotive design. In PBD, design is carried out as the mapping of a user-defined function to a platform instance. This platform instance represents a network of interconnected components, chosen from a library. Together with their functionality, in PBD components expose also other characteristics such as composition rules, performance indices, and cost. This additional information is used to optimize the mapping process, according to both functional and non-functional specifications.

In this paper, we borrow from PBD the idea that platform components can define their own composition rules. Moreover, every component exposes some non-functional characteristics (expressed by means of a cost function), which need to be optimized during the mapping process.

3. Preliminaries on A/G Contracts

Contracts provide a formal framework to instantiate Platform-Based Design, defining rules for correct-by-construction component integration and validation, as well as concepts of refinement and abstraction [23–26].

Assume/Guarantee contracts represent a specific instance of the contract theory, where the behavior of a component, i.e., its promise or guarantee, and its expectations from the environment, i.e., its assumption, are expressed using assertions. Here, A/G contracts are defined using synchronous assertions, which are sets of behaviors. A behavior is a sequence of evaluations of variables from a fixed alphabet Σ_{IO} sharing the same domain.

Formally, an A/G contract is a pair $\varphi = (A, G)$ where A and G are synchronous assertions representing assumptions and guarantees, respectively. Any environment behavior $E \subseteq A$ is a valid behavior for φ, while any component behavior M such that $M \cap A \subseteq G$ is a behavior which satisfies φ, i.e., the component is behaving correctly under the assumptions of the contract. If $A = \emptyset$, the contract is said *incompatible*, i.e., there is no valid environment for it, and if $G = \emptyset$, it is *inconsistent*, i.e., there is no valid implementation for it.

A contract is *saturated* if its guarantees are maximal, i.e., $G \supseteq \overline{A}$, where $\overline{A}$ is the complement of A[1]. In the rest of the paper, even if not explicitly stated,

[1]Note that the contracts $\varphi_1 = (A, G)$ and $\varphi_2 = (A, G \cup \overline{A})$ are equivalent, as every behavior

we will always refer to saturated contracts.

We concretely express the sets A and G as a pair of LTL formulas, ψ_A and ψ_G, each denoting the set of all traces (behaviors) that satisfy it. As for A, and G, also ψ_A and ψ_G are defined over the same set of variables Σ_{IO}.

Contract algebra includes a number of operations to manipulate contracts, including parallel *composition* and *refinement*, which are required by the problem that we will illustrate in Def. 4.1. The parallel composition of two contracts $\varphi_1 = (\psi_{A1}, \psi_{G1})$ and $\varphi_2 = (\psi_{A2}, \psi_{G2})$ can be directly defined in terms of LTL formulas as

$$\varphi_1 \otimes \varphi_2 = ((\psi_{A1} \wedge \psi_{A2}) \vee \neg(\psi_{G1} \wedge \psi_{G2}), \psi_{G1} \wedge \psi_{G2}). \tag{1}$$

Contract composition is associative and commutative, and preserves the saturated form; if φ_1 and φ_2 are saturated, then so is $\varphi_1 \otimes \varphi_2$.

Refinement, instead, formalizes a notion of substitutability and it is defined as a preorder on contracts. A contract $\varphi_1 = (\psi_{A1}, \psi_{G1})$ refines contract $\varphi_2 = (\psi_{A2}, \psi_{G2})$, written $\varphi_1 \preceq \varphi_2$, if $\psi_{A2} \Rightarrow \psi_{A1}$ and $\psi_{G1} \Rightarrow \psi_{G2}$ are both valid or, equivalently, if $\neg(\psi_{A2} \Rightarrow \psi_{A1})$ and $\neg(\psi_{A1} \Rightarrow \psi_{G2})$ are both unsatisfiable. Two contracts φ_1 and φ_2 are said *equivalent* if and only if $\varphi_1 \preceq \varphi_2$ and $\varphi_2 \preceq \varphi_1$. Refinement can be efficiently verified, for LTL A/G contracts, using any tool able to check satisfiability of LTL formulas, such as a model checker.

In Section 6, we discuss the implementation of our algorithm which uses components specified as LTL A/G contracts. Nevertheless, our solution is general and works with other compositional frameworks as well.

4. Constrained synthesis from component libraries (CSCL)

In our framework, a *component* $G \in \mathbb{G}$, where $\mathbb{G}$ is the domain representing the space of all possible components, is a tuple $G = (I_G, O_G, \varphi_G, \sigma_G, R_G)$. I_G is the set of input ports, O_G is the set of output ports, and φ_G is the component specification, expressed using a specific notation (e.g., an A/G contract, or an LTL formula). Variables in φ_G correspond to ports in I_G and O_G. I_G, O_G, and φ_G are all defined over a common set of symbols, or alphabet, Σ_{IO}. The function $\sigma_G : I_G \cup O_G \rightarrow T$ maps ports of G to elements in T, where T is a *typeset*. A typeset is a poset consisting of a set of symbols (types) ordered by the *subtype* relation[2]. For $a, b \in T$, the notation $a \leq b$ means that b is a subtype of a. Finally, R_G is a set of logic constraints over ports of G.

A library is a tuple $L = (Z, T, R_Z, f)$. Here, $Z = \{G_1, \dots, G_n\}$ is a finite set of components. Several components in Z can have the same specification φ_G, but they are required to have at least unique names for ports (and variables). R_Z

M that satisfies φ_1 also satisfies φ_2, and *vice versa*.

[2]Without loss of generality, here we can consider the poset T being organized as a *tree*. This is enough to obtain a simple type system with single inheritance, where all the types share the same *root type* ($\perp$). The choice of T, however, does not have an impact on the general formulation of our framework.

is a set of logic constraints that encode connection rules over ports of components in Z and types in T. Constraints in R_G and R_Z characterize a certain library, and are used by the library designer to provide domain-specific insights that can be used to speed up the synthesis process. The cost function $f : \wp(\mathbb{G}) \to \mathbb{R}$, where $\wp(\mathbb{G})$ is the powerset of $\mathbb{G}$, maps sets of components from the domain $\mathbb{G}$ to real numbers, i.e., the cost of the component.

The use of a cost function associated to the library derives from Platform-Based Design principles, that is, components not only need to satisfy a specification, but they can also expose non-functional characteristics which need to be optimized during instantiation. These non-functional characteristics of components are captured in our framework using f. Here, f is defined with the library, as different problem domains have different notions of cost.

We consider the system specification $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$, that needs to be synthesized, as a component itself. In this way (through constraints in R_S) a user of the synthesizer is also able to provide design hints that are specific to the problem instance, such as input/output interface (in terms of ports and their types) as well as additional constraints over those ports.

Later on, for a library $L = (Z, T, R_Z, f)$, we will use the following set to address all the ports of all components in it:

$$\mathcal{P}_L = \bigcup_{i=1}^{|Z|} I_{G1} \cup O_{G1} \tag{2}$$

Similarly, to indicate all the ports in L *and* the ports in the system specification S, we use:

$$\mathcal{P}_{L \cup S} = \mathcal{P}_L \cup I_S \cup O_S \tag{3}$$

The composition of two components $G_1 = (I_1, O_1, \varphi_1, \sigma_1, R_1)$ and $G_2 = (I_2, O_2, \varphi_2, \sigma_2, R_2)$ is a new component $G_1 \otimes G_2 = ((I_1 \cup I_2) \backslash (O_1 \cup O_2), O_1 \cup O_2, \varphi_1 \otimes \varphi_2, \sigma_1 \cup \sigma_2, R_1 \cup R_2)$, assuming that the operator $\otimes$ is defined for φ_1 and φ_2. This means that input ports that are connected to output ports are considered outputs in the resulting composition. For instance, when input a is connected to output b, the resulting composite component only contains output b (input a "disappears" since it is going to be controlled by b). We also assume that there is no conflict between σ_1 and σ_2, meaning that ports with the same name need to have the same type according to both σ_1 and σ_2:

$$\forall p \in I_1 \cup O_1 : p \in I_2 \cup O_2 \Rightarrow \sigma_1(p) = \sigma_2(p)$$

We say that a component G_1 refines a component G_2, written $G_1 \preceq G_2$, if and only if

$$I_1 \subseteq I_2, O_2 \subseteq O_1, \text{ and } \varphi_1 \preceq \varphi_2 \tag{4}$$

where we assume that the formalism used to express component specifications φ_1 and φ_2 includes the notion of refinement. For instance, if φ_1 and φ_2 are logic formulas, $\varphi_1 \preceq \varphi_2$ is equivalent to the implication $\varphi_1 \Rightarrow \varphi_2$. Intuitively, if G_1

refines G_2, then φ_2 will always hold if φ_1 holds, i.e., G_1 can be safely used in place of G_2.

To describe the logic constraints in R_Z, R_G, and R_S, and to model interactions between components (e.g., when the output of a component is the input of another one) we need to introduce the concept of connection between ports. Formally, we indicate connections between ports using the function:

$$\rho : \Sigma_{IO} \times \Sigma_{IO} \rightarrow \{0, 1\} \tag{5}$$

Given components G_1 and G_2 with ports p and q, respectively, we have $\rho(p, q) = 1$ [3] if and only if, in the resulting composition $(G_1 \otimes G_2)$ or refinement operation $(G_1 \preceq G_2)$, the variables corresponding to p and q in φ_{G1} and φ_{G2} will be expressed (i.e. renamed) using a new, unused common symbol:

$$\rho_{p,q} \Leftrightarrow \exists \text{ fresh } x \in \Sigma_{IO} : [p \rightsquigarrow x]_{\varphi_{G1}} \wedge [q \rightsquigarrow x]_{\varphi_{G2}} \tag{6}$$

where the notation $[p \rightsquigarrow x]_{\varphi_{G1}}$ indicates the renaming of variable p with x within the formula φ_{G1}. A variable $x \in \Sigma_{IO}$ is *fresh* if and only if $x \notin \mathcal{P}_{L \cup S}$.

Now let us also consider a component G_3 containing a port t. If a port is connected multiple times to different ports, then all of their corresponding variables will be renamed using the same new symbol:

$$\rho_{p,q} \wedge \rho_{p,t} \Leftrightarrow \exists \text{ fresh } x \in \Sigma_{IO} : [p \rightsquigarrow x]_{\varphi_{G1}} \wedge [q \rightsquigarrow x]_{\varphi_{G2}} \wedge [t \rightsquigarrow x]_{\varphi_{G3}} \tag{7}$$

Note that using ρ to define connections between components can, potentially, yield inconsistent renaming of variables and thus inconsistent compositions of components. Consider, for instance, three ports p, q, t and a function ρ such that $\rho_{p,q}$, $\rho_{p,t}$, and $\neg\rho_{q,t}$. Clearly, no such renaming could be applied because the first two connections imply $\rho_{q,t}$. In Section 4.2, we describe how to properly constrain ρ to avoid such situations.

Often we will refer to the composition of components (and their specifications) using the notation $G_1 \otimes_\rho G_2$, where with $\otimes_\rho$ we indicate the renaming of variables in φ_{G1} and φ_{G2} according to ρ, followed by the composition operation defined above. Similarly, $G_1 \preceq_\rho G_2$ will indicate a renaming of variables followed by the refinement operation. Two components G_1 and G_2 are *equivalent* if and only if there exists a renaming function ρ such that $G_1 \preceq_\rho G_2$ and $G_2 \preceq_\rho G_1$.

Example 1 (Component Connection)**.** Let

$$G_1 = (I_1, O_1, \varphi_1, \sigma_1, R_1) = (\{a_1, b_1\}, \{c_1\}, c_1 = a_1 + b_1, \{a_1, b_1, c_1\} \rightarrow \{\bot\}, \emptyset)$$
$$G_2 = (I_2, O_2, \varphi_2, \sigma_2, R_2) = (\{a_2\}, \{b_2\}, b_2 = 2 \cdot a_2, \{a_2, b_2\} \rightarrow \{\bot\}, \emptyset)$$

be two components and ρ a renaming function specifying a single connection ρ_{b_1,b_2} (thus $\neg\rho_{a_1,b_2}, \neg\rho_{c_1,b_2}, \neg\rho_{a_1,a_2}$, etc.). Let us also assume that component

[3] To indicate $\rho(p, q) = 1$, we will often use the shorthand $\rho_{p,q}$, and $\neg\rho_{p,q}$ for $\rho(p, q) = 0$.

specifications can be composed by taking their conjunction. Then, the composition $G_1 \otimes_\rho G_2$ yields a component

$$G_1 \otimes_\rho G_2 = (\{a_1, a_2\}, \{c_1, b_3\}, b_3 = 2 \cdot a_2 \wedge c_1 = a_1 + b_3, \{a_1, a_2, b_3, c_1\} \to \{\perp\}, \emptyset)$$

Where b_3 is a fresh symbol replacing b_1 and b_2.

Example 2 (Running example: synthesize the modulo operation)**.** We introduce here a simple example to help the reader familiarize with the concepts introduced so far. Our objective is to synthesize the modulo operation starting from a library of simpler arithmetic operations. For simplicity, we assume only strictly positive integer inputs.

Let us define our library to be $L_{op} = (Z_{op}, \{\perp\}, \emptyset, f(G) = 0)$, where we have only one type ($\perp$) for all the ports and no additional constraints over ports and types. $Z_{op} = \{\mathit{add}, \mathit{sub}, \mathit{mult}, \mathit{div}\}$ is a set containing addition, subtraction, multiplication and integer division, and $f(G) = 0$ is a constant function.

Every component has two inputs and one output, and its specification is the associated arithmetic operation. We assume no additional constraints over ports also at component level. Thus we have:

$$\begin{aligned}
&\mathit{add} = (\{a_a, b_a\}, \{c_a\}, c_a = a_a + b_a, \{a_a, b_a, c_a\} \to \{\perp\}, \emptyset) \\
&\mathit{sub} = (\{a_s, b_s\}, \{c_s\}, c_s = a_s - b_s, \{a_s, b_s, c_s\} \to \{\perp\}, \emptyset) \\
&\mathit{mult} = (\{a_m, b_m\}, \{c_m\}, c_m = a_m \cdot b_m, \{a_m, b_m, c_m\} \to \{\perp\}, \emptyset) \\
&\mathit{div} = (\{a_d, b_d\}, \{c_d\}, c_d = \lfloor a_d / b_d \rfloor, \{a_d, b_d, c_d\} \to \{\perp\}, \emptyset)
\end{aligned}$$

To successfully find a solution, we need to make sure the operations of composition and refinement are defined for elements in L_{op}. Here, the composition of two component specifications is the classical *function composition*, while the refinement relation can simply be the equivalence between functions.

The specification is the component $S_{mod} = (\{x, y\}, \{z\}, z = \mathit{mod}(x, y), \{x, y, z\} \to \{\perp\}, \emptyset)$. We know that the modulo operation can be computed as $\mathit{mod}(x, y) = x - \lfloor x/y \rfloor \cdot y$. A composition of elements in Z_{op} that implements S_{mod} is shown in Figure 1: $\mathit{sub}(x, \mathit{mult}(\mathit{div}(x, y), y))$, with connections ρ_{b_s, c_m}, ρ_{a_m, c_d}, ρ_{x, a_s}, ρ_{x, a_d}, ρ_{y, b_d}, ρ_{y, b_m}, ρ_{z, c_s}.

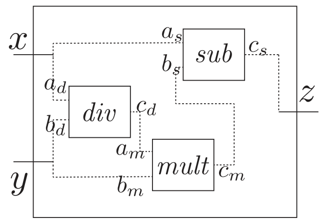


Figure 1: Modulo operation composition from elements in L_{op}.

In the following sections we will define a set of rules to automatically obtain candidate solutions which are topologically sound (e.g., adding the connection

ρ_{z,c_m} should be illegal because the output z is already connected, or controlled, by the port c_s), and semantically correct (e.g., having ρ_{z,c_m} *instead of* ρ_{z,c_s} would yield a composition which does not implement the modulo operation, although topologically sound).

4.1. A combinatorial analysis of the CSCL problem

The problem of composing a finite number of elements from a library is hard. In this section, we quantify its combinatorial complexity by analyzing two simpler cases first and then putting the results together for the general case. As in the previous section, we consider a library $L = (Z, T, R_Z, R_T, f)$, with finite $Z = \{G_1, \dots, G_n\}$, and a specification $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$. Since we are interested in the worst-case scenario, in this case we assume $R_Z = R_T = R_S = R_{G_1} = \cdots = R_{G_n} = \emptyset$, and $T = \{\bot\}$ (a typeset containing only the root type).

First, we examine the case in which we already have a set of m *connected* components $H = \{G'_1, \dots, G'_m\}$ and we want to find a single component $G_z \in Z$ such that $\varphi_z \otimes \varphi'_1 \otimes \cdots \otimes \varphi'_m \preceq \varphi_S$. Assuming n is the number of components in Z, we have n possibilities to try. Extending this example to include $c \leq n$ unknown components is straightforward. In this case, there are $\frac{n!}{c!(n-c)!}$ possible solutions, assuming that the order of components does not matter.

On the other hand, we have a scenario in which we still have m components $H = \{G'_1, \dots, G'_m\}$, but connections among them are missing. We want to connect the components to each other, according to a certain function ρ, such that $\varphi'_1 \otimes_\rho \cdots \otimes_\rho \varphi'_m \preceq \varphi_S$. The complexity of this problem depends on the total number of ports. Assuming p is the number of ports of a component, then there are $2^{\frac{mp(mp-1)}{2}}$ possible solutions.[4]

Combining together the previous two examples yields the worst case for the CSCL scenario, in which we want to find both components and their connections to satisfy S. Assuming every component in our library Z has at most p ports, and a finite N as the maximum number of components in a possible solution, one can see how in this case there are $\Sigma_{c=1}^{N} \frac{n!}{c!(n-c)!} 2^{\frac{cp(cp-1)}{2}}$ possible solutions.

4.2. Synthesis Constraints

The analysis in Section 4.1 shows that the CSCL problem grows quickly with the number of components and ports in the library. The role of the library-specific constraints is to mitigate such complexity. We require R_Z to contain *at least* the constraints defined in the following paragraphs[5]:

[4]Recall that the maximum number of edges in a graph of n nodes is $\frac{n(n-1)}{2}$. Then, $2^{\frac{n(n-1)}{2}}$ is the number of all the subsets of those connections.

[5]Here we borrow the notation typical of *first-order* logic formulas, although all the formulas refer to a finite number of elements.

- Connections must be consistent, according to the following properties which encode the semantics of ρ. Equation 8 tells us that if for three ports p, q, w we have $\rho_{p,q}$ and $\rho_{q,w}$, then it must be also $\rho_{p,w}$:

$$\forall p, q, w \in \mathcal{P}_{L\cup S} : \rho_{p,q} \wedge \rho_{q,w} \Rightarrow \rho_{p,w} \tag{8}$$

  Equation 9 represents the fact that if p is connected to q, then q is also connected to p:

$$\forall p, q \in \mathcal{P}_{L\cup S} : \rho_{p,q} \Rightarrow \rho_{q,p} \tag{9}$$

  Equation 10 simply states that a port is always connected to itself:

$$\forall p \in \mathcal{P}_{L\cup S} : \rho_{p,p} \tag{10}$$

- Two output ports of two different components in the library cannot be connected to each other:

$$\forall G, G' \in Z : \forall p, q \in O_G \cup O_{G'} : (p \neq q) \Rightarrow \neg\rho_{p,q} \tag{11}$$

- Components representing a candidate solution are collected in the set $H \subseteq Z$, with maximum size N. Inputs of a component in H must be connected either to inputs of S or outputs of other components in H:

$$\forall G \in H : \forall p \in I_G : (\exists s \in I_S : \rho_{p,s}) \vee (\exists G' \in H : \exists q \in O_{G'} : \rho_{p,q}) \tag{12}$$

  Example 3. Equation 11 prevents the connection between multiple outputs of components in H. With respect to Ex. 2, this means enforcing $\neg\rho_{c_a,c_s}$, $\neg\rho_{c_a,c_m}$, $\neg\rho_{c_a,c_d}$, $\neg\rho_{c_s,c_m}$, $\neg\rho_{c_s,c_d}$, and $\neg\rho_{c_m,c_d}$. Equation 12, instead, makes sure that no inputs of the components in H are left unconnected. For instance, the composition in Figure 2 violates Equation 12, because a_s is not connected to any other port.

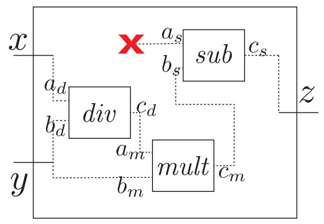


Figure 2: Illegal composition of elements in L_{op} (a_s disconnected).

- No distinct ports of S can be connected to each other. In this case, such constraint is not too restrictive. If needed, in fact, one can relax this constraint by explicitly adding a component in the library implementing the identity function:

$$\forall s, r \in I_S \cup O_S : s \neq r \Rightarrow \neg\rho_{s,r} \tag{13}$$

- Inputs of the specification S cannot be connected to component outputs, because otherwise in the resulting composition those inputs will be treated as outputs (as seen in Section 4):

$$\forall s \in I_S : \forall G \in Z : \forall p \in O_G : \neg \rho_{s,p} \tag{14}$$

- Every input of the specification S has to be connected at least to an input of a component in H (Equation 15), while every output of S has to be connected at least to an output of a component in H (Equation 16):

$$\forall s \in I_S : \exists G \in H : \exists p \in I_G : \rho_{s,p} \tag{15}$$

$$\forall s \in O_S : \exists G \in H : \exists p \in O_G : \rho_{s,p} \tag{16}$$

Example 4. Equation 15 and Equation 16 ensure that there is a full mapping of specification ports into components ports. For instance, the composition in Figure 3 violates Equation 16 because there is an output of the specification, z, which is not connected to any component outputs.

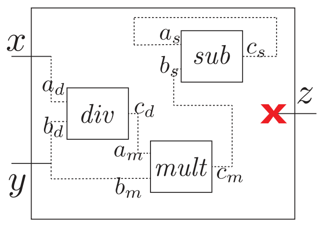


Figure 3: Illegal composition of elements in L_{op} (z disconnected).

- Only ports with compatible types can be connected to each other, according to the subtype relation defined in Section 4 and considering contravariant inputs and outputs. This means that, given two ports p and q connected to each other, if p is an output and q is an input, then $\sigma(p) \leq \sigma(q)$, and *vice versa* (similarly, in principle, to what is described by de Alfaro and Henzinger in [27]):

$$\forall G, G' \in Z : \forall p \in I_G, q \in O_{G'} : \sigma_G(p) \not\leq \sigma_{G'}(p) \Rightarrow \neg \rho_{p,q} \tag{17}$$

4.2.1. Problem Definition

The following definitions introduce the problem of Constrained Synthesis from Component Libraries (CSCL). Our goal is to describe the problem in a way that is as general as possible. We achieve this by:

1. only requiring components to be defined using a formalism that provides the operations of composition and refinement. Contracts, logic formulas, and finite state machines to name a few, all satisfy this condition.

2. requiring the library of components to satisfy *at least* the constraints presented in Equations 8 to 17, allowing the designer, however, to add as many constraints as necessary, according to the problem domain.

Definition 4.1 (CSCL problem)**.** Let $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$ be a system specification, and $L = (Z, T, R_Z, f)$ a library of components where R_Z contains at least the constraints described in Equations 8 to 17. Let also the operations of composition ($\otimes$) and refinement ($\preceq$) be defined for components in L. The problem of Constrained Synthesis from Component Libraries consists of finding a finite set of components $H = \{G_1, \dots, G_N \mid G_i = (I_i, O_i, \varphi_i, \sigma_i, R_{Gi}) \in Z\}$ and a connection function ρ such that the cost function f is minimized according to:

$$\begin{aligned} \underset{\{G_1,\dots,G_N\}}{\text{minimize}} \quad & f(\{G_1, \dots, G_N\}) && \text{(18a)}\\ \text{subject to} \quad & \text{all synthesis constraints in } R_Z,\ R_S,\ R_{Gi}\ , && \text{(18b)}\\ & \varphi_1 \otimes_\rho \cdots \otimes_\rho \varphi_N \preceq \varphi_S && \text{(18c)} \end{aligned}$$

In case the function f is a constant, then the CSCL problem can be simplified as follows.

Definition 4.2 (Simplified CSCL problem)**.** Let $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$ be a system specification, and $L = (Z, T, R_Z, f)$ a library of components where R_Z contains at least the constraints described in Equations 8 to 17, and f is a constant. Let also the operations of composition ($\otimes$) and refinement ($\preceq$) be defined for components in L. The simplified CSCL problem consists of finding a finite set of components $H = \{G_1, \dots, G_N \mid G_i = (I_i, O_i, \varphi_i, \sigma_i, R_{Gi}) \in Z\}$ and a connection function ρ such that:

$$\begin{aligned} \text{all synthesis constraints in } R_Z,\ R_S,\ R_{Gi} \text{ hold} && \text{(19a)}\\ \varphi_1 \otimes_\rho \cdots \otimes_\rho \varphi_N \preceq \varphi_S && \text{(19b)} \end{aligned}$$

5. Solving a concrete instance of the CSCL problem

The CSCL problem in Def. 4.1 (and 4.2) is very general, and the most effective approach to solve it depends on the structure of the library. For instance, a continuous cost function f will require optimization techniques which are very different from a cost function which is purely discrete, or which depends on the formalism used to describe component specifications.

In this section we discuss a solution based on the following assumptions:

- f can be solved using discrete optimization techniques;

- f does not depend on the formalism used to describe the specification of components. This means that, for a component G, $f(G)$ can be evaluated without considering φ_G.

This choice allows us to effectively decouple the topological aspects of a candidate solution of the CSCL problem in Def. 4.1, i.e., Equation 18a and 18b, from its semantic evaluation, i.e., Equation 18c.

Under these assumptions, we propose a solution based on the CEGIS paradigm, in which synthesis is carried out by an iterative algorithm. In each iteration two major steps are performed:

STEP 1 A discrete optimization problem is solved to retrieve a candidate solution, that is, a set of components, and their connections, which minimizes the objective function and satisfies all the synthesis constraints. With respect to Def. 4.1, this first step takes care of Equation 18a and 18b, and provides the function ρ used to solve Equation 18c. This step, in general, can be solved by a constraint solver. With respect to Example 2, for instance, this step corresponds to the generation of possible solution candidates such as $\mathit{sub}(x, \mathit{mult}(\mathit{div}(x, y), y))$, or $\mathit{add}(x, \mathit{mult}(\mathit{div}(x, y), y))$.

STEP 2 Equation 18c is checked by interrogating a tool, which we call *verifier*, able to *understand* component specifications. The verifier determines whether the candidate composition, after proper interconnection of components, refines the global specification φ_S. In Example 2, the verifier is the tool able to determine, indeed, that $\mathit{sub}(x, \mathit{mult}(\mathit{div}(x, y), y))$ implements the modulo operation.

The choice of the verifier used in the second step depends on the formalism used to specify components. For instance, a model checker could be chosen as verifier in case components are specified as state machines and the global specification is an LTL formula or, in case of ordinary differential equations, a numerical solver. In this paper, both the system specification and the components are described using LTL A/G contracts.

We call *counterexample* a candidate composition (i.e., a set of components and their connections) which has been proven wrong by the verifier. A counterexample is used to *inductively* learn new constraints for the solver. In general, the performance of a CEGIS-based algorithm depends on how well a counterexample can prune the search space, leveraging as much as possible the information that can be inferred from the components specifications and execution traces, if the verifier provides them. In this work, however, we make no assumption on the verifier which is used to check the validity of candidates. Our solution leverages component equivalence, introduced in Section 4. We can do so by using the refinement operation that we assume in Definitions 4.1 and 4.2, thus preserving the generality of the approach.

We can identify equivalent components using the function

$$E : \mathbb{G} \to 2^{\mathbb{G} \times \Sigma_{IO}^{\Sigma_{IO}}} \tag{20}$$

which takes a component as input and returns a set of pairs, consisting of a component and a function mapping ports to ports. Given a component G in a library L, $E(G)$ returns a set containing all the pairs (G', M) such that there exists a renaming function that makes G' equivalent to G. M, then, maps the ports of G to the ports of G' according to such renaming function. Formally,

$$E(G) = \left\{ (G', M) \;\middle|\; \begin{array}{l} G' \in Z \text{ and} \\ \exists \rho' : \\ \quad G \preceq_{\rho'} G' \text{ and } G' \preceq_{\rho'} G \text{ and} \\ \quad \forall p \in I \cup O \quad : \\ \quad \forall p' \in I' \cup O' \;\; : \\ \qquad\qquad \rho'_{p,p'} \Leftrightarrow M(p) = p' \end{array} \right\} \tag{21}$$

In general, E is fixed for a given library and can easily be precomputed and accessed during synthesis, without significant performance overhead.

5.1. The CSCL *algorithm*

As mentioned earlier in this section, the assumption that makes the application of our algorithm possible is that the cost function f does not depend on the formalism used to describe the specifications of components, and that f can be minimized using discrete optimization techniques. In this way, there is a clear separation between the satisfaction of the synthesis constraints, including the minimization of the cost function, and the evaluation of the refinement relation between the system specification and the composition of components.

A number of constraint solvers are able, indeed, to minimize objective functions while satisfying logic constraints, such as Z3 [28]. The simpler formulation of the CSCL problems in Def. 4.2, on the other hand, doesn't require the function f to be minimized and can be solved by the CSCL algorithm using a constraint solver without optimization capabilities.

In the CSCL algorithm, illustrated in Table 1, the task of pruning the search space is carried out in a twofold manner. First, the constraint solver only needs to search over a number of potential candidates which is drastically limited by the synthesis constraints. Such constraints include those encoded in the library through the constraints in R_Z, $R_{G_1}, \ldots, R_{G_N}$, and R_S.

Second, each time a counterexample is observed in step 3.1, it is used to match all the elements of the library equivalent to those in the counterexample, according to the component equivalent sets described in Equation 21, and the algorithm in Table 2. This allows us to rule out a number of possible candidate instances exponential in the number of components contained in the rejected candidate. For instance, if the counterexample is a composition of 4 components, and for each one there are 2 other components in Z with the same specification, then adding constraints from that single counterexample will discard 3^4 erroneous candidate instances.

The output of the CSCL algorithm is a finite set of components, H, and their connections, expressed as a function, ρ. To ensure termination, the CSCL algorithm requires a bound on the number of components used in a candidate

Algorithm 1: CSCL

Input: A specification $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$, a library of components $L = (Z, T, R_Z, f)$, the maximum number of components in the solution N.
Output: A set of components $H = \{G_1, \ldots, G_n\}, H \subseteq Z$ and $n \leq N$, and a connection function ρ, such that $\varphi_1 \otimes_\rho \cdots \otimes_\rho \varphi_n \preceq \varphi_S$ and f in minimized, or `NULL` if no solution is found.

```
1. Initialize constraint solver and verifier, instantiating the synthesis constraints
   for problem instance and setting $N$ as the maximum number of components in
   a candidate solution.
2. (STEP 1) WHILE get candidate solution $(H' = \{G_1, \ldots, G_n\}, \rho')$, with $n \leq N$,
   from constraint solver such that $f$ is minimized and all the synthesis constraint
   hold:
    1. Build composition $G_1 \otimes_{\rho'} \cdots \otimes_{\rho'} G_n$.
    2. (STEP 2) IF the verifier checks that $\varphi_1 \otimes_{\rho'} \cdots \otimes_{\rho'} \varphi_n \preceq \varphi_S$ holds:
           RETURN $H = H'$ and $\rho = \rho'$.
    3. ELSE
        1. infer incorrect renamings from candidate $(H', \rho')$:
           $R = \textsc{RejectCandidate}(H', \rho')$.
        2. FOR ALL $\rho_{\text{temp}} \in R$:
            1. add constraint $\rho \neq \rho_{\text{temp}}$ to constraint solver.
3. return NULL.
```

Table 1: CSCL algorithm. (STEP 1) and (STEP 2) are labels. The algorithm interfaces with a constraint solver (preferably with optimization capabilities) to execute (STEP 1), and with a verifier, e.g., a model checker, to execute (STEP 2).

solution. The choice of such bound depends on the details of the problem being solved. In the CSCL algorithm, we let the user decide what is the most appropriate bound through the input parameter N.

The complexity of the CSCL algorithm depends on the structure of the library and the solution maximum size. According to our analysis in Section 4.1, for a fixed library, the worst case time complexity will be exponential in the maximum number of components in a solution, N, multiplied by the complexity of the call to the verifier. In practice, though, the techniques discussed above are able to limit the search space in a considerable manner, yielding acceptable synthesis times in many cases.

5.2. An efficient encoding for the synthesis constraints

The encoding of the synthesis constraints presented in Section 4.2 is not particularly efficient. For instance, one can see how Equation 8 represents a formula which grows cubically in the number of ports of all components in the library. Such encoding would cause the internal representation of the synthesis constraints in the constraint solver to grow unnecessarily large, resulting in

Algorithm 2: RejectCandidate

Input: A set of contracts $H = \{G_1, \ldots, G_n\}$, and a renaming function ρ.
Output: R, a set containing renaming functions

```
1. $R = \text{set}()$
2. FOR ALL $G_1, G_2 \in H$:
    1. Initialize $\rho_{\text{temp}} = \rho$
    2. FOR ALL $(G'_1, M_1) \in E(G_1)$:
        1. FOR ALL $(G'_2, M_2) \in E(G_2)$:
            1. FOR ALL $p_1 \in I_1 \cup O_1$:
                1. FOR ALL $p_2 \in I_2 \cup O_2$:
                    1. Update $\rho_{\text{temp}}(M_1(p_1), M_2(p_2)) = \rho(p_1, p_2)$.
    3. Add $\rho_{\text{temp}}$ to $R$.
3. RETURN $R$
```

Table 2: RejectCandidate algorithm. The returned set R collects all the possible renaming functions that would yield a candidate equivalent to (H, ρ).

poor performance. In this section, we present an encoding that exploits a more efficient representation of component connections.

Given library $L = (Z, T, R_Z, f)$ and system specification $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$, we assign an *index* to all the output ports in the library and also to all the input ports of the specification S, and indicate with $\mathbb{I}$ the set containing such indices. Conversely, we associate an integer variable to every input port in the library, as well as to every output port of S. We call these variables *connection variables* and group them in the set $\mathbb{V}$. Connection variables, as the name suggests, are used to specify connections between ports, and they are *assigned* by the constraint solver in the first step of the CSCL algorithm. We use the function $\mathcal{I} : \mathcal{P}_{L \cup S} \to \mathbb{I} \cup \{-1\}$ to retrieve the index of a given port, or -1 if the index is not defined for the port. Similarly, we use the function $\mathcal{V} : \mathcal{P}_{L \cup S} \to \mathbb{V} \cup \{\emptyset\}$ to retrieve the connection variable of a given port, or $\emptyset$ if the connection variable is not defined for that port (e.g., input ports of S). We then map connection variables to indices using the function $\mathcal{M} : \mathbb{V} \to \mathbb{I} \cup \{-1\}$. Such encoding allows us to eliminate the expensive explicit representation of function ρ. For instance, the new encoding will represent the assertion $\rho_{p,q}$ for an input port p and output port q with the assignment $\mathcal{M}(\mathcal{V}(p)) = \mathcal{I}(q)$, also indicated, for convenience, as $\mathcal{M}(p) = \mathcal{I}(q)$. If p is not connected to any port, then $\mathcal{M}(p) = -1$. We only allow inputs from the library to be connected to outputs in the library or inputs of the specification S:

$$\forall G \in Z : \forall p \in I_G : \\ (\mathcal{M}(p) = -1) \vee [\exists G' \in Z : \exists q \in O_{G'} : \mathcal{M}(p) = \mathcal{I}(q)] \vee [\exists s \in I_S : \mathcal{M}(p) = \mathcal{I}(s)] \quad (22)$$

We also impose that outputs of the specification S can only be mapped to outputs from the library:

$$\forall s \in O_S : (\mathcal{M}(s) = -1) \vee (\exists G \in Z : \exists p \in O_G : \mathcal{M}(s) = \mathcal{I}(p)) \tag{23}$$

The following theorem states that using the encoding presented in this section, with Equations 23 and 22, yields a solution space which is at least as large as the one obtained representing ρ using Equations 8 to 17. Our goal is to show that if a solution exists within the constraint in Equations 8 to 17, then it will exist also using the encoding introduced here.

Theorem 5.1. *Let $\mathbf{C_1}$ be the set of connections among components in $Z \cup \{S\}$ that can be defined by the function ρ according to Equations 8 to 17. Let also $\mathbf{C_2}$ be the set of connections that can be defined by the connection variables in $\mathbb{V}$ and indices in $\mathbb{I}$, constrained by Equations 22 and 23. Then $\mathbf{C_2} \subseteq \mathbf{C_1}$.*

Proof. We start considering only connections between ports in $\mathcal{P}_L$. Given an input port p and an output port q, a connection $\rho_{p,q}$ in $\mathbf{C_1}$ (and by Equation 9 also $\rho_{q,p}$) can be trivially be represented in $\mathbf{C_2}$ by the assignment $\mathcal{M}(p) = \mathcal{I}(q)$. If both p and q are outputs, then by Equation 11 their connection cannot be in $\mathbf{C_1}$. If both p and q are inputs and $\rho_{p,q}$ is in $\mathbf{C_1}$, then by Equation 12 p and q have to be connected to another output in the library or to an input of S. In either case, assume w be such port, where $\rho_{p,w}$ and $\rho_{q,w}$ are also in $\mathbf{C_1}$. Then $\mathcal{M}(p) = \mathcal{I}(w)$ and $\mathcal{M}(q) = \mathcal{I}(w)$ represent the equivalent connections in $\mathbf{C_2}$, including indirectly $\rho_{p,q}$ (because they have a reference to the same index). Consider now also ports of the specification S. Since, in $\mathbf{C_1}$, we do not allow any two ports of S being connected to each other (Equation 13), we have only the case in which there is a connection $\rho_{s,p}$ between ports $s \in I_S \cup O_S$ and $p \in \mathcal{P}_L$. If s is an input, then p has to be an input too (because of Equation 14), and we can represent $\rho_{s,p}$ as $\mathcal{M}(p) = \mathcal{I}(s)$ in $\mathbf{C_2}$. If s is an output, then p can be either a component input or output. If p is an output, then $\rho_{s,p}$ can be represented as $\mathcal{M}(s) = \mathcal{I}(p)$. If p is an input, then by Equation 16 there must be another component output q such that $\rho_{s,q}$. By Equation 8, then it must be also $\rho_{p,q}$. Therefore we can map these three connections in $\mathbf{C_2}$ with $\mathcal{M}(s) = \mathcal{I}(q)$ and $\mathcal{M}(p) = \mathcal{I}(q)$ (where $\rho_{s,q}$ is implicit because s and p refer to the same index). This shows that all the connections in $\mathbf{C_1}$ have an equivalent in $\mathbf{C_2}$, hence $\mathbf{C_1} \subseteq \mathbf{C_2}$. $\square$

Thanks to Theorem 5.1, we are ensured that the encoding presented here (under Equations 22 and 23) preserves the solution space defined by Equations 8 to 17. All the results in Section 7 are obtained after reformulating the synthesis constraints in Equations 8 to 17 using the encoding described in this section.

6. Implementing the CSCL algorithm

In this and the following sections, we describe the implementation of a parallel variant of the CSCL algorithm and evaluate its capabilities and performance.

We used the SMT solver *Z3* [28] to find candidates satisfying the synthesis constraints and minimize the cost function f, and we chose to represent our library as a set of LTL A/G contracts. This choice is also motivated by the fact that composition and refinement operations are well defined in the contract algebra. Moreover, additional concepts such as compatibility and consistency can be leveraged to derive, before the actual synthesis process, library constraints on components composability (in the form of incompatible sets of ports stored through constraints in R_Z). Lastly, but not less important, several tools are available to deal with LTL specifications. In our experiments, the verifier chosen to compute refinement checks is *NuXMV* [29].

An efficient implementation of the CSCL algorithm has been developed using the encoding described in Section 5.2. Additionally, we decided to modify the CSCL algorithm to exploit multiprocessor architectures and further speed up synthesis. The CSCL algorithm, in Table 1, first computes a candidate solution and then it asks the verifier to validate or discard that candidate. The verifier execution is, in general, a time-consuming operation, i.e., verifying the validity of an LTL formula is a PSPACE-complete problem [30, 31]. We can observe, however, that it is possible (and convenient) to interrogate several verifier instances at the same time, providing them with different candidates. Here, we modify CSCL algorithm following this intuition, i.e., rejecting a candidate as soon as it is given to the verifier, and providing the ability to retrieve an *old* candidate in case one of the many verifier instances gives a positive answer. Table 3 illustrates the parallel version of the CSCL algorithm in Table 1. The Parallel CSCL algorithm is equivalent to the CSCL algorithm, as the two algorithms perform the same operations, with the difference that the Parallel CSCL generates candidates continuously, stopping only when a certain verifier instance indicates a successful candidate. The implementation of Parallel CSCL resulted in a tool we call pyco[6].

To evaluate a candidate solution in H, pyco considers three possible cost functions, defined as follows:

- Minimize the number of components in H:

$$f(H) = |H|$$

- Minimize the number of ports used in the solution:

$$f(H) = \sum_{h \in H} |I_h| + |O_h|$$

- Minimize a user-defined cost, based on the cost c of each component in the library:

$$f(H) = \sum_{h \in H} c_h$$

[6] pyco, together with all the experiments discussed in this paper, is available at `https://github.com/ianno/pyco/releases/tag/SCP2018`

Algorithm 3: PARALLEL CSCL

Input: A specification $S = (I_S, O_S, \varphi_S, \sigma_S, R_S)$, a library of components $L = (Z, T, R_Z, f)$, the maximum number of components in the solution N.

Output: A set of components $H = \{G_1, \ldots, G_n\}, H \subseteq Z$ and $n \leq N$, and a connection function ρ, such that $\varphi_1 \otimes_\rho \cdots \otimes_\rho \varphi_n \preceq \varphi_S$ and f in minimized, or `NULL` if no solution is found.

```
1. Initialize constraint solver and verifier, instantiating synthesis constraints for
   problem instance and setting $N$ as the maximum number of components in a
   candidate solution.
2. WHILE get candidate solution $(H' = \{G_1, \ldots, G_n\}, \rho')$ from constraint solver
   such that $f$ is minimized and all the synthesis constraint hold:
    1. Build composition $G_1 \otimes_{\rho'} \cdots \otimes_{\rho'} G_n$.
    2. Spawn a new verifier instance (process) to verify $\varphi_1 \otimes_{\rho'} \cdots \otimes_{\rho'} \varphi_n \preceq \varphi_S$.
    3. IF any verifier instance has signaled success:
           retrieve instance and RETURN $H = H'$ and $\rho = \rho'$.
    4. ELSE
        1. infer incorrect renamings from candidate $(H', \rho')$:
           $R = \text{RejectCandidate}(H', \rho')$.
        2. FOR ALL $\rho_{\text{temp}} \in R$:
            1. add constraint $\rho \neq \rho_{\text{temp}}$ to constraint solver.
3. Wait for all the remaining running verifier instances to terminate.
4. IF any verifier instance has signaled success:
       retrieve instance and return $H = H'$ and $\rho = \rho'$.
5. return NULL.
```

Table 3: PARALLEL CSCL algorithm.

7. Case studies

This section contains two examples which illustrate the capability of the tool we developed, and, in general, of the CSCL algorithm. The goal of the first example is to design the control logic of a Brushless DC electric Motor (BLDC) Driver. The specification describes the waveform of the current used to control the electromagnets of the motor, and the task of the synthesizer is to figure out what components are necessary and how to connect them to ensure its proper operation. The second example, more challenging, requires the synthesizer to provide the control logic for the controller of an aircraft Electrical Power System (EPS), where the specification describes some strict safety requirements that need to be always satisfied.

In our experiments, given a certain specification, we observed that synthesis time behaves like a heavy-tailed random distribution, making it hard to provide an accurate estimation of its mean value. We justify this behavior by observing that SMT solvers, such as Z3, have intrinsically non-deterministic performance. Thus, the search space is explored in a slightly different manner each time an experiment is executed. To characterize the mean, we decided to run each experiment 100 times, as we observed that this number is a good compromise between total computation time and quality of the sample space, and then bootstrapped our data to compute the 95% confidence interval of the mean synthesis time.

We ran all the experiments on a 3.3 GHz Intel Xeon machine, with 32GB of RAM, limiting the maximum number of parallel processes to 8. In some cases, however, we ran some experiments using the single process CSCL algorithm in Table 1.

7.1. The Brushless DC electric Motor Design (BLDC)

Typical DC electric motors have permanent magnets which are fixed (*stator*), containing a spinning armature (*rotor*). The armature contains an electromagnet that, when powered, attracts some magnets in the stator and repels others, causing a partial rotation of the rotor. To keep the rotation going, it is necessary to periodically invert the polarity of the electromagnet. This task is executed by some metal brushes on the rotor that, making contact with the electrodes on the stator, flip the polarity of the electromagnet as they rotate. This design is simple but presents a number of limitations, such as the physical wear of brushes and low performance.

Brushless DC electric motors, as the name suggests, overcome those limitations by not having rotating brushes. In these motors, often the electromagnets are located on the stator while magnets are on the rotor, and the change in polarity is handled by a computer through high-power transistors [32]. BLDC motors are more precise, efficient, and have better performance than regular DC motors. They are more complex, however, as they require more electronic components to work properly. Although BLDC motors can be one, two, or three-phase, most of them are usually the latter type.

One of the simplest motor drivers for three-phase BLDC motors is the so-called *half bridge* configuration. Half bridges have this name because they only support the positive polarization of the electromagnets (instead of both negative and positive), generating only half of the maximum torque. Figure 4 shows the typical half bridge driver configuration, while Figure 5 illustrates the current waveforms required for the proper operation of the motor.

The motor driver needs to properly open and close the half-bridge switches (i.e., transistors), reading the current position of the rotor provided by a *Hall effect* sensor placed in its proximity. Once the rotor reaches a commutation point, the sensor sends an impulse to the driver, which takes care of actuating the switches.

The goal of this example is to synthesize an architecture of components which is able to drive a simple BDCL motor. In doing so, we want to show how our tool is able to infer a number of necessary components, correctly satisfying

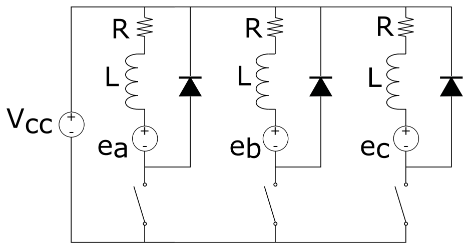


Figure 4: Common BLDC half bridge motor driver topology. Variables e_a, e_b, and e_c represent the Electromotive Force (EMF) for the three phases of the motor [32].

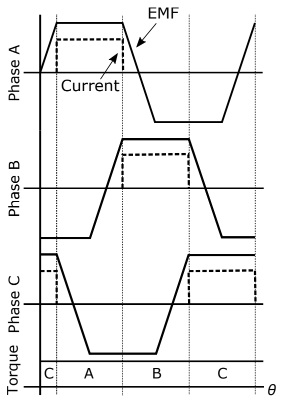


Figure 5: Waveforms for the half bridge driver. Input current from the driver induces torque through one of the three phases of the motor at a time [32]. To ensure the proper forward rotation of the rotor, inputs from the driver need to be sent in a specific order.

the specification both semantically (i.e., the A/G contract of the composition refines the A/G contract of the specification), and topologically (i.e., all the port types match). In this example, all the variables used in A/G contracts are Boolean. Table 4 illustrates the specification used for this case study.

The specification describes the interface of the motor driver, including one input and three outputs, at a logic level, without taking into account other physical constraints. The input, i, communicates to the driver whether the motor has reached a commutation point. The three outputs are signals which drive the current of the three phases of the motor. Given the input, the specification only requires that, when a commutation point is detected, only one driver signal is sent to the motor. The task of the synthesizer is to choose components from the library of 18 components, described in Table 5, and properly connect them to satisfy the specification.

Most of the components in the library only expose their interface, without specifying any logic. For instance, the component *Power-12V* is a power generator which only provides ports for ground and voltage. The *MCU* component, on the other hand, already has the control logic required to satisfy the specifica-

Input Ports	i	(IOPin3V)
Output Ports	o_1, o_2, o_3	(IOPin12V)
Assumptions	$\neg i \land \Box\Diamond i \land \Box\Diamond\neg i$	
Guarantees	$o_1 \land \neg o_2 \land \neg o_3 \land$ $\Box[(o_1 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc\neg o_1 \land \bigcirc o_2 \land \bigcirc\neg o_3)]\land$ $\Box[(o_2 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc\neg o_1 \land \bigcirc\neg o_2 \land \bigcirc o_3)]\land$ $\Box[(o_3 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc o_1 \land \bigcirc\neg o_2 \land \bigcirc\neg o_3)]$	
$\mathbf{R_S}$	Distinct(o_1, o_2, o_3)	

Table 4: Specification for the BLDC synthesis problem. The interface has one input, which is a 3V pin from the Hall effect sensor (its type is *IOPin3*), and three outputs as 12V pins to drive the electromagnets of the motor. The specification assumes that the input is initially negative and, once started, it will keep commuting. The guarantee is that only one output line will be active at each commutation point in a round-robin fashion. The specification also requires distinct outputs, meaning that they cannot be controlled by the same port.

Component	Input Ports	Output Ports	Assumptions	Guarantees
Power-5V	- -	*gnd* (GND) *vout* (Voltage5V)	true	true
DCDC-3V	*gnd* (GND) *vin* (Voltage12V)	*vout* (Voltage3V)	true	true
DCDC-5V	*gnd* (GND) *vin* (Voltage12V)	*vout* (Voltage5V)	true	true
Power-12V	- -	*gnd* (GND) *vout* (Voltage12V)	true	true
MCU	*gnd* (GND) *vin* (Voltage3V) *i* (IOPin3V)	o_1 (IOPin3V) o_2 (IOPin3V) o_3 (IOPin3V)	true	$o_1 \land \neg o_2 \land \neg o_3 \land$ $\Box[(o_1 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc\neg o_1 \land \bigcirc o_2 \land \bigcirc\neg o_3)]\land$ $\Box[(o_1 \land \neg i \land \bigcirc\neg i)\Rightarrow (\bigcirc o_1 \land \bigcirc\neg o_2 \land \bigcirc\neg o_3)]\land$ $\Box[(o_2 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc\neg o_1 \land \bigcirc\neg o_2 \land \bigcirc o_3)]\land$ $\Box[(o_2 \land \neg i \land \bigcirc\neg i)\Rightarrow (\bigcirc\neg o_1 \land \bigcirc o_2 \land \bigcirc\neg o_3)]\land$ $\Box[(o_3 \land \neg i \land \bigcirc i) \Rightarrow (\bigcirc o_1 \land \bigcirc\neg o_2 \land \bigcirc\neg o_3)]\land$ $\Box[(o_3 \land \neg i \land \bigcirc\neg i)\Rightarrow (\bigcirc\neg o_1 \land \bigcirc\neg o_2 \land \bigcirc o_3)]$
Half-Bridge	*gnd* (GND) *vin* (Voltage3V) *i* (IOPin3V)	*o* (IOPin12V)	true	$\Box(i = o)$
$\mathbf{R_T}$	$\emptyset$		$\mathbf{R_Z}$	$\emptyset$

Table 5: Structure of the BLDC library, which contains three separate instances of each component. Some components are architectural, meaning that they provide a typed interface but their A/G contract is always satisfied, and some are logic, providing both a typed interface and a non-trivial A/G contract (such as the MCU).

tion. Their port types, however, mismatch. The outputs of *MCU*, indeed, have type *IOPin3V*, while the specification requires outputs with type *IOPin12V*. Thus, Equation17 prevents their direct connection. It is responsibility of the synthesizer to figure out the right connections to propagate the control logic to ports of the right type.

We asked the synthesizer to find a solution using a constant cost function, minimizing the number of components in the solution, and minimizing the total number of ports. Additionally, we also ran a series of experiments using the single process CSCL algorithm in Table 1, with a constant cost function. For each case, we ran 100 experiments summarized in Figure 6.

Interestingly, the overall fastest set of experiments was the one in which we minimized the number of components. Conversely, minimizing the number of ports led to the slowest synthesis times. Although the performance of constraints solvers is, in general, non-deterministic, we can explain these results by

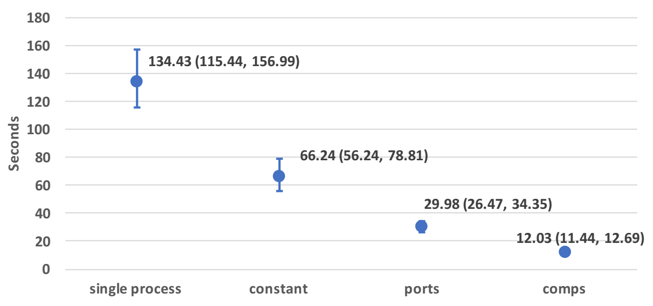


Figure 6: Summary of the results for the BLDC experiments. We synthesized a controller using the single process CSCL algorithm first, with a constant cost function. Then, we used the PARALLEL CSCL algorithm with a constant cost function, minimizing the number of components, and minimizing the number of ports. For each category, we ran 100 experiments. Each point represents the mean synthesis time, while the bars represent its 95% confidence interval, also indicated within parentheses.

observing that the MCU component, although necessary to satisfy the specification, is the one with most ports. To minimize the number of ports used, the synthesizer tries avoiding it, without success, leading to a longer synthesis time. All the experiments resulted in correct designs, where the typical configuration was the set of components {Power-12V, DCDC-3V, MCU, Half-Bridge, Half-Bridge1, Half-Bridge2}, correctly connected. Figure 7 illustrates the typical solution for this case study. In the figure, the arrows between ports represent the function ρ, defining port connections. An arrow between two ports, say a and b, means $\rho(a, b) = 1$.

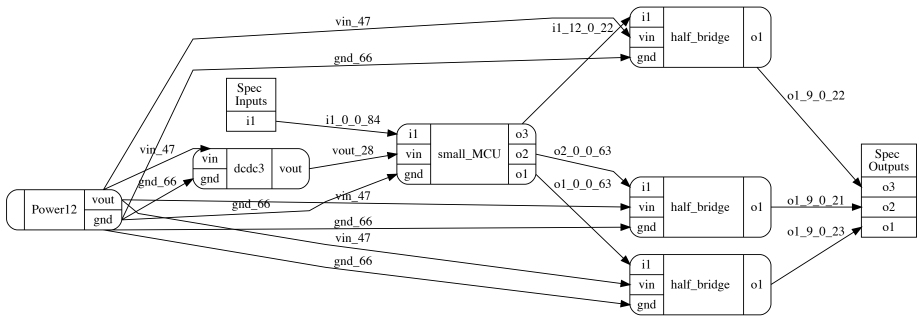


Figure 7: Graphical representation of a synthesized design. Names on arrows represent port renamings induced by the function ρ, returned by the synthesis process.

7.2. The Aircraft Electrical Power System (EPS)

Figure 8 shows the simplified structure of an aircraft EPS in the form of a *single-line diagram*[7] [9–11]. Generators (as those on the top left and right sides of the diagram) deliver power to the loads (e.g., avionics, lighting, heating, and motors) via AC and DC buses. In the event of generator failures, *Auxiliary Power Units* (APUs) will provide the required power. Some buses supply loads which are critical, therefore they cannot be unpowered for more than a predefined amount of time. Other, non-essential, buses supply loads that may be shed in the case of a fault. The power flow from sources to loads is determined by contactors, which are electromechanical switches that can be opened or closed. *Transformer Rectifier Units* (TRUs) convert and route AC power to DC buses.

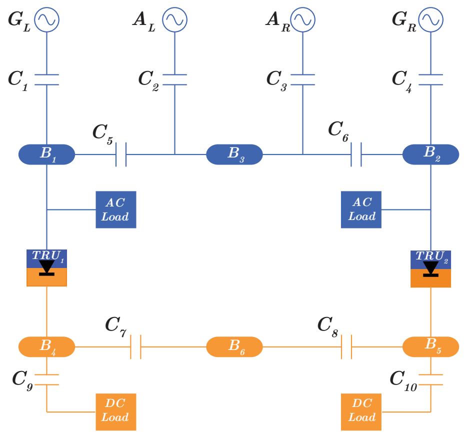


Figure 8: Single line diagram of the EPS.

The function of the controller, called *Bus Power Control Unit* (BPCU), is to react to changes in system conditions or failures and reroute power by actuating the contactors, ensuring that essential buses are adequately powered. Generators, APUs, and TRUs are components subject to failures.

Our goal is to synthesize the logic of the BCPU from a set of subsystem controllers, described by a library of A/G contracts. In our model, controller inputs are expressed as Boolean variables, corresponding to the state of the various physical elements (i.e., presence or absence of faults). Controller outputs are also described using Boolean variables and represent the status of the contactors in the system (open or closed). At this level of abstraction, contactors are assumed to have a negligible reaction time.

Table 6 illustrates the set of specifications that the BPCU needs to satisfy. The first two rows on the left describe what are the input and output ports of the EPS plant and their types, indicated in parenthesis next to the port names

[7]Single line diagrams are usually used to simplify the description of three-phase power systems.

Input Ports	G_L, G_R	(ActiveGenerator)
	A_L, A_R	(BackupGenerator)
	R_L, R_R	(Rectifier)
Output Ports	C_1, C_4	(ACGenContactor)
	C_2, C_3	(ACGenContactor)
	C_5, C_6	(ACBackContactor)
	C_7, C_8	(DCBackContactor)
	C_9, C_{10}	(DCLoadContactor)
Assumptions (common to all)	$\neg G_L \wedge \Box(G_L \Rightarrow \bigcirc G_L)\wedge$	
	$\neg G_R \wedge \Box(G_R \Rightarrow \bigcirc G_R)\wedge$	
	$\neg A_L \wedge \Box(A_L \Rightarrow \bigcirc A_L)\wedge$	
	$\neg A_R \wedge \Box(A_R \Rightarrow \bigcirc A_R)\wedge$	
	$\neg R_L \wedge \Box(R_L \Rightarrow \bigcirc R_L)\wedge$	
	$\neg R_R \wedge \Box(R_R \Rightarrow \bigcirc R_R)$	

$\boldsymbol{S_1}$	$C_1 \wedge \Box(G_L \Rightarrow \bigcirc\neg C_1)$
$\boldsymbol{S_2}$	$C_4 \wedge \Box(G_R \Rightarrow \bigcirc\neg C_4)$
$\boldsymbol{S_3}$	$\Box(A_L \Rightarrow \bigcirc\neg C_2)$
$\boldsymbol{S_4}$	$\Box(A_R \Rightarrow \bigcirc\neg C_3)$
$\boldsymbol{S_5}$	$\Box\neg(C_2 \wedge C_3)$
$\boldsymbol{S_6}$	$\Box[(\neg G_L \wedge \neg G_R) \Rightarrow \Diamond\neg(C_5 \wedge C_6)]$
$\boldsymbol{S_7}$	$\Box[(\neg G_L \wedge \neg A_L \wedge \neg A_R \wedge \neg G_R) \Rightarrow$ $\Diamond(\neg C_2 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_6)]$
$\boldsymbol{S_8}$	$\Box[\neg(R_L \wedge R_R) \Rightarrow C_9]$
$\boldsymbol{S_9}$	$\Box[\neg(R_L \wedge R_R) \Rightarrow C_{10}]$
$\mathbf{R_S}$	Distinct output ports

Table 6: Set of system specifications $S_1 \dots S_9$ to satisfy. Input ports reflect the status of EPS elements (such as generators), while output ports represent contactors. Assumptions are common to all the specifications and capture the expectation that when a component fails, it will not be operational again. Guarantees include the promise that faulty generators will be isolated, no short-circuit will happen, and loads will always be powered. The specifications also require distinct outputs, meaning that each of them has to be controlled by a separate port.

(Figure 9 shows the type tree associated with ports in the specification and library components). In total, each specification is defined over 6 input and 10 output ports.

Input ports $G_L, G_R, A_L, A_R, R_L, R_R$ represent the environment event of failure of the left and right generator, APU, and TRU, respectively. Output ports

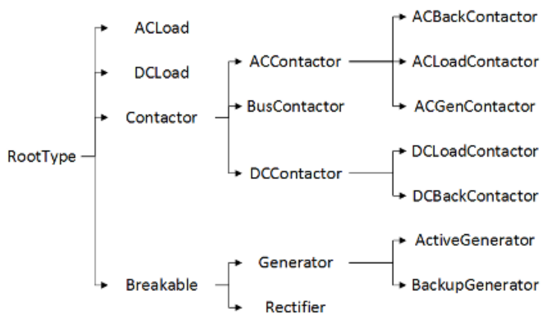


Figure 9: Tree representing the typeset used in the EPS case study.

$C_1, \dots, C_{10}$ represent the state of the contactors. The second column of Table 6 describes a set of 9 specifications, all sharing the same assumptions. In this example, we assume from the environment that all the components do not start to operate in a faulty state (see, for instance, $\neg G_L$ in the first line of the assumptions in Table 6, referring to the left generator), and if a component breaks, then it will stay broken (specified, for the left generator, by $\Box(G_L \Rightarrow \bigcirc G_L)$). Specifications S_1 to S_4 require that if a generator or APU breaks, then it will be disconnected from the rest of the EPS in the next execution step. Note that

S_1 and S_2 require also the two generators to be initially connected to the rest of the plant. S_5 requires the absence of a short circuit between the two APUs, while S_6 requires the absence of a short circuit between generators in case they are both healthy (after an initial setup period). Furthermore, S_7 specifies that bus B_3 needs to be isolated if no faults in generators or APUs occur. Finally, S_8 and S_9 require that DC loads need to be connected to the plant if at least one TRU is working correctly. In this example, the synthesis constraints include also the restriction that the variables associated to the specifications cannot be connected to each other, i.e., the failure of two EPS components needs to be associated to distinct events. These hints are encoded as a relation in R_S.

Table 7 shows the components and the user-defined constraints (in this example only type compatibility) in the library. Every component is described by its I/O ports (annotated with their types), and its specification as an A/G pair. All the components make some assumptions over the state of a certain type of EPS elements and provide a guarantee over the state of some contactors. Consider, for instance, component B_1. It just assumes that a certain generator is not initially broken (note that the type of the input variable allows it to be connected to either a generator or an APU), and guarantees that the contactor will be always open. Clearly, B_1 is not a good candidate to satisfy either S_1 or S_2, since they require the contactor to be closed at least initially. Similarly, all the other components in the library encode a particular behavior that can be used to control parts of the EPS.

We ran two series of experiments, one using a library containing 20 components, and one with 40 components. In both cases, the goal was to synthesize the BPCU according to the specifications in Table 6. For each series, we asked the synthesizer to find a solution first using the single process CSLC algorithm in Table 1 with a constant cost function, and then using the parallel version in Table 3 with a constant cost function minimizing the number of components used, and minimizing the number of ports in the solution. With both libraries, trying to minimize the number of components in the solution led to very long synthesis times, beyond the timeout that we set at 200 seconds.

This is, indeed, one of the main risks in trying to synthesize a composition minimizing a cost function; depending on the distribution of the solutions in the search space, the synthesizer might spend a lot of time exploring a set of candidates with low cost, but far from any useful solution.

Figure 10 shows the observed results, in terms of execution time, for the remaining cases. Interestingly, in the case of minimization of the number of ports, the synthesizer was able to find a solution generally faster than in the case of a constant cost function. In each graph, the dot represents the average synthesis time for one of the specification subsets $\{S_1\}, \{S_1, S_2\}, \ldots, \{S_1, \ldots, S_9\}$. For each of these subsets, we ran 100 experiments reporting their mean values, together with their 95% confidence intervals, in Table 8. As expected, one can immediately see how the parallel approach to synthesis is indeed more efficient than the single process one, with up to 50% performance improvement.

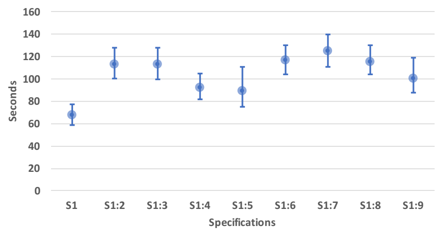
For reference, a typical solution satisfying all 9 specifications with the minimal number of connected ports included 5 components, $\{I_1, D_1, L_1, G_1, D_2\}$, for

Comp.	Input Ports	Output Ports	Assumptions	Guarantees
$\mathbf{A_1}$	f (Generator)	c (ACGenContactor)	$\neg f \wedge$ $\square(f \Rightarrow \bigcirc f)$	$\square(f \Rightarrow \Diamond\neg c)$
$\mathbf{B_1}$	f (Generator)	c (ACGenContactor)	$\neg f$	$\square(\neg c)$
$\mathbf{C_1}$	f (ActiveGenerator)	c (ACGenContactor)	$\neg f \wedge$ $\square(f \Rightarrow \bigcirc f)$	$\square(f \Rightarrow \neg c) \wedge$ $\square(\neg f \Rightarrow c)$
$\mathbf{D_1}$	f (ActiveGenerator)	c (ACGenContactor)	$\neg f \wedge$ $\square(f \Rightarrow \bigcirc f)$	$c \wedge$ $\square(f \Rightarrow \bigcirc\neg c) \wedge$ $\square(\neg f \Rightarrow c)$
$\mathbf{E_1}$	f_1 (Generator) f_2 (Generator)	c (ACBackContactor)	$\neg f_1 \wedge \neg f_2 \wedge$ $\square(f_1 \Rightarrow \bigcirc f_1) \wedge$ $\square(f_2 \Rightarrow \bigcirc f_2)$	$\square((f_1 \vee f_2) \Rightarrow c) \wedge$ $\square((\neg f_1 \wedge \neg f_2) \Rightarrow \neg c)$
$\mathbf{F_1}$	f_1 (BackupGenerator) f_2 (BackupGenerator)	c_1 (ACGenContactor) c_2 (ACGenContactor)	$\neg f_1 \wedge \neg f_2 \wedge$ $\square(f_1 \Rightarrow \bigcirc f_1) \wedge$ $\square(f_2 \Rightarrow \bigcirc f_2)$	$\square[(\neg f_1 \wedge \neg f_2) \Rightarrow$ $(\neg c_1 \wedge \neg c_2)] \wedge$ $\square[(f_1 \wedge \neg f_2) \Rightarrow$ $(\neg c_1 \wedge \neg c_2)] \wedge$ $\square[(\neg f_1 \wedge f_2) \Rightarrow$ $(c_1 \wedge c_2)] \wedge$ $\square[(f_1 \wedge f_2) \Rightarrow$ $(\neg c_1 \wedge c_2)]$
$\mathbf{G_1}$	f_1 (ActiveGenerator) f_4 (ActiveGenerator) f_2 (BackupGenerator) f_3 (BackupGenerator)	c_1 (ACBackContactor) c_4 (ACBackContactor) c_2 (ACGenContactor) c_3 (ACGenContactor)	$\neg f_1 \wedge \neg f_2 \wedge \neg f_3 \wedge \neg f_4 \wedge$ $\square(f_1 \Rightarrow \bigcirc f_1) \wedge$ $\square(f_2 \Rightarrow \bigcirc f_2) \wedge$ $\square(f_3 \Rightarrow \bigcirc f_3) \wedge$ $\square(f_4 \Rightarrow \bigcirc f_4)$	$\square(f_2 \Rightarrow \neg c_2) \wedge$ $\square(f_3 \Rightarrow \neg c_3) \wedge$ $\square(\neg(c_2 \wedge c_3)) \wedge$ $\square[(\neg f_1 \wedge \neg f_4) \Rightarrow$ $(\neg c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \neg c_4)] \wedge$ $\square[(\neg f_1 \wedge \neg f_3 \wedge f_4) \Rightarrow$ $(\neg c_1 \wedge \neg c_2 \wedge c_3 \wedge c_4)] \wedge$ $\square[(f_1 \wedge \neg f_2 \wedge \neg f_4) \Rightarrow$ $(c_1 \wedge c_2 \wedge \neg c_3 \wedge \neg c_4)] \wedge$ $\square[(\neg f_1 \wedge \neg f_2 \wedge f_3 \wedge f_4) \Rightarrow$ $(\neg c_1 \wedge c_2 \wedge \neg c_3 \wedge c_4)] \wedge$ $\square[(f_1 \wedge f_2 \wedge \neg f_3 \wedge \neg f_4) \Rightarrow$ $(c_1 \wedge \neg c_2 \wedge c_3 \wedge \neg c_4)] \wedge$ $\square[(f_2 \wedge f_3 \wedge (f_1 \vee f_4)) \Rightarrow$ $(c_1 \wedge \neg c_2 \wedge c_3 \wedge c_4)]$
$\mathbf{H_1}$	f (Rectifier)	c (ACLoadContactor)	$\neg f$	$\square(\neg f \Rightarrow c) \wedge$ $\square(f \Rightarrow \neg c)$
$\mathbf{I_1}$	f_1 (Rectifier) f_2 (Rectifier)	c_1 (DCBackContactor) c_2 (DCBackContactor)	$\neg f_1 \wedge \neg f_2$	$\square[(\neg f_1 \wedge \neg f_2) \Rightarrow$ $(\neg c_1 \wedge \neg c_2)] \wedge$ $\square[(f_1 \vee f_2) \Rightarrow$ $(c_1 \wedge c_2)]$
$\mathbf{L_1}$	f_1 (Rectifier) f_2 (Rectifier)	c (DCLoadContactor)	$\neg f_1 \wedge \neg f_2$	$\square c$
$\mathbf{R_T}$	{(Generator, ACGenContactor)}		$\mathbf{R_Z}$	$\emptyset$

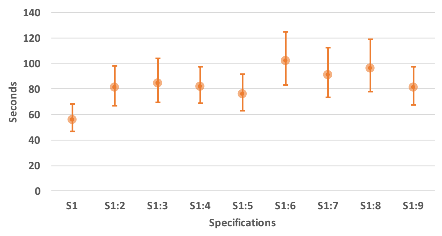
Table 7: Structure of the EPS library. In our experiments, the library contained first 2 and then 4 instances of these components, for a total of 20 and 40 elements.

	Single Process 20	Constant Cost 20	Minimize Ports 20	Single Process 40	Constant Cost 40	Minimize Ports 40
$\{S_1\}$	67.43 (58.49, 76.95)	34.69 (30.85, 38.44)	10.69 (10.42, 10.98)	55.92 (46.44, 67.97)	28.13 (24.25, 34.40)	31.17 (30.07, 32.42)
$\{S_1, S_2\}$	113.12 (99.99, 127.91)	62.71 (54.84, 73.55)	15.48 (14.88, 16.16)	81.18 (67.07, 98.20)	48.29 (41.30, 57.93)	40.90 (38.94, 43.45)
$\{S_1, \ldots, S_3\}$	112.86 (99.69, 128.07)	61.93 (54.26, 72.62)	17.21 (16.59, 17.87)	84.67 (69.63, 104.13)	58.18 (49.58, 68.20)	47.87 (45.38, 50.70)
$\{S_1, \ldots, S_4\}$	92.21 (81.43, 104.75)	49.10 (43.31, 57.24)	17.94 (17.25, 18.84)	81.94 (68.54, 97.43)	49.78 (43.45, 57.83)	50.90 (49.12, 52.75)
$\{S_1, \ldots, S_5\}$	89.31 (75.28, 110.72)	41.31 (36.40, 48.44)	16.55 (16.08, 17.10)	76.27 (63.28, 91.91)	43.58 (37.62, 51.67)	53.20 (51.35, 55.76)
$\{S_1, \ldots, S_6\}$	116.42 (103.59, 129.76)	60.96 (54.48, 69.29)	18.91 (18.41, 19.44)	101.86 (82.91, 124.56)	50.64 (43.50, 60.69)	63.30 (61.23, 65.68)
$\{S_1, \ldots, S_7\}$	124.48 (110.38, 139.84)	66.87 (59.03, 78.11)	20.86 (20.19, 21.69)	90.70 (73.34, 112.62)	69.78 (59.88, 81.82)	66.28 (64.24, 68.75)
$\{S_1, \ldots, S_8\}$	115.44 (103.73, 129.81)	64.71 (57.64, 74.75)	22.72 (21.98, 23.70)	96.10 (77.74, 119.08)	56.19 (48.33, 66.11)	71.20 (69.16, 73.59)
$\{S_1, \ldots, S_9\}$	100.54 (87.27, 118.88)	42.83 (38.78, 47.58)	22.26 (21.78, 22.79)	80.91 (67.39, 97.48)	64.09 (55.57, 74.49)	75.61 (73.78, 77.51)

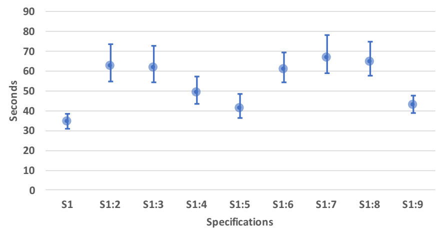
Table 8: Summary of the EPS experiments. For each specification subset (one for each row), we report the mean value and its 95% confidence interval. All values are expressed in seconds. Experiments named "Constant Cost" and "Minimize Ports" are run using parallel processes.



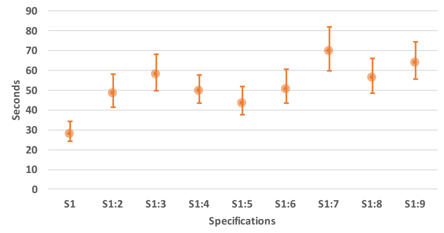
(a) Single process, constant cost, 20 elements.



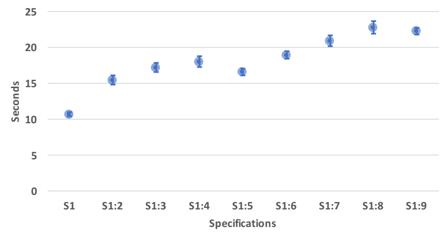
(b) Single process, constant cost, 40 elements.



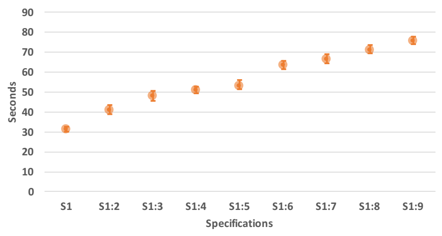
(c) Parallel execution, constant cost, 20 elements.



(d) Parallel execution, constant cost, 40 elements.



(e) Parallel execution, minimize ports, 20 elements.



(f) Parallel execution, minimize ports, 40 elements.

Figure 10: BPCU synthesis times in the various experiments. In each graph, each point has been computed running 100 experiments. The central point represents the mean, while bars represent the 95% confidence interval for the mean. The horizontal axis refers to the subset of specifications considered in each experiment.

a total of 19 ports connected accordingly. Figure 11 represents the connections among the components according to the renaming function ρ.

In a separate experiment, using the library with 40 elements, PYCO was able to explore the whole search space invoking the verifier 108176 times. This corresponded to more than 400M rejected candidates, which did not require an explicit check thanks to the inductive learning process described in Section 5. The verifier found a solution satisfying the specifications 386 times, corresponding to roughly 1.5M equivalent ones in the search space.

Figure 12 shows, instead, the effect of designer hints and library-specific constraints on synthesis time. Here synthesis is performed on smaller and sim-

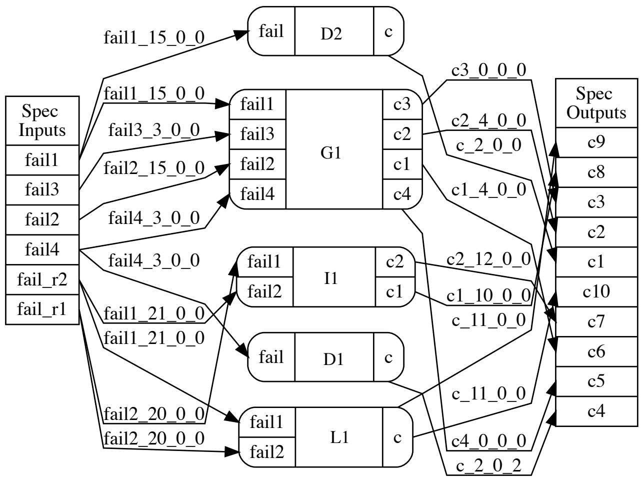


Figure 11: Graphical representation of a synthesized design satisfying all 9 specifications with minimal number of connected ports. Names on arrows represent port renamings induced by the function ρ, returned by the synthesis process.

plified instances of the EPS problem, including 2, 4, 6, 10 and 16 ports, and using a library with 20 elements. The graph (in logarithmic scale), shows how these constraints are critical in decreasing the overall problem complexity. In case of the instance with 16 ports, the CSCL algorithm variant without types and additional constraints was not able to synthesize a solution within our 1000 seconds timeout.

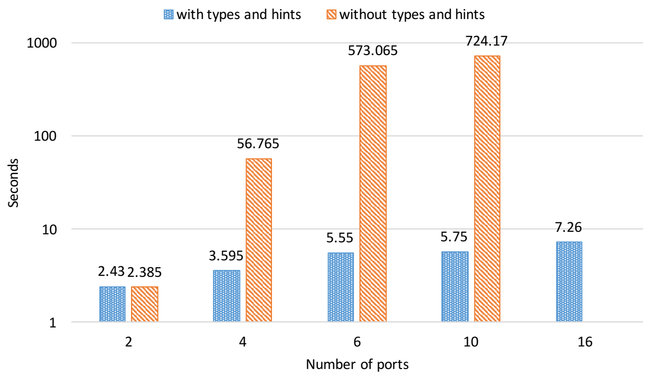


Figure 12: Impact of types and user provided hints on synthesis time for simplified instances of the EPS example. Each bar of the histogram represents the median value from 10 experiments. The graph is in logarithmic scale. In the case without types and 16 ports, most of the times the synthesizer has not been able to find a solution within the time limit of 1000 seconds.

8. Discussion

The work described in this paper focuses on a very specific aspect of CPS design, which is the problem of synthesizing a controller by composing components defined at a certain level of abstraction. The design of CPS, however, is a very complex task and we believe it can be fully automated only using a combination of techniques, allowing the designer to manage the process at different levels of abstraction, from system requirements to physical details.

Ideally, each component in a design library would expose a number of interfaces, consistent with each other, enabling the iterative mapping process which is typical of Platform-Based Design. For instance, the theory of design contracts [23] introduces the notion of *viewpoints*, which can be used to describe different component aspects (e.g., functional, timing, etc.) and perfectly fit our idea of design. Always in [23], Benveniste *et al.* describe an abstraction framework for contracts based on the notion of *Galois connection*. Contract abstractions are compositional with respect to parallel composition and conjunction. They also allow to disprove refinement, and check consistency and compatibility of concrete contracts. Thus, this framework represents a fundamental step in defining a flexible, yet formal, approach to design where it is possible to seamlessly reason between contracts and components at different abstraction levels.

In the context of the work presented in this paper, one can imagine each component defined through an LTL A/G contract providing a corresponding implementation, i.e., a state machine, which exposes more concrete properties. This is not unreasonable, as each component implementation could be easily verified for correctness using a model checker, or even automatically synthesized using the techniques discussed in Section 2. We are releasing the software that implements the algorithms discussed in this article as we believe it can be used as a platform for future work.

9. Conclusion

In this paper, we studied the problem of constrained synthesis from component libraries. After defining the general theoretical framework, and assessing the complexity of the domain, we have proposed a problem formulation in terms of generic components subject to a cost function and a number of synthesis constraints. These constraints include types on component ports, suggestions from the designer, and composition rules which can also be precomputed and stored in the library.

We presented two variants of an algorithm based on CEGIS, a sequential and a parallel one, and evaluated its implementation with LTL A/G contracts on industrial-relevant case studies. Analyzing the case studies, we believe that the potential of our approach has emerged clearly, although some criticalities, such as the heavy impact that the choice of cost function can have on synthesis times, are still challenging.

Future extensions of this work include the study of algorithms to decompose complex specifications into smaller instances (to increase performance by dealing with smaller synthesis problems), the application of the synthesis technique described here to component libraries defined over multi-aspect specifications (e.g., behavioral, security-related, real-time), and the analysis of erroneous designs and infeasible specifications in order to provide feedback to the designer on how to fix her library and obtain the intended result.

Acknowledgments

The authors wish to acknowledge Christos Stergiou, Sanjit Seshia, Richard Lin, Rohit Ramesh, and the anonymous reviewers for the useful comments. This work has been partially supported by the NSF (CCF-1139138, CNS-1329759 and CNS-1801546), by the Camozzi Group via the iCyPhy consortium, by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by the Academy of Finland.

[1] A. Pnueli, R. Rosner, Distributed reactive systems are hard to synthesize, in: Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on, 1990, pp. 746–757 vol.2. `doi:10.1109/FSCS.1990.89597`.

[2] Y. Lustig, M. Y. Vardi, Synthesis from component libraries, in: Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, FOSSACS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 395–409. `doi:10.1007/978-3-642-00596-1_28`.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, SIGOPS Oper. Syst. Rev. 40 (5) (2006) 404–415. `doi:10.1145/1168917.1168907`.

[4] T. Wongpiromsarn, U. Topcu, N. Ozay, H. Xu, R. M. Murray, Tulip: A software toolbox for receding horizon temporal logic planning, in: Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC '11, ACM, New York, NY, USA, 2011, pp. 313–314. `doi:10.1145/1967701.1967747`.

[5] S. Jha, S. Gulwani, S. A. Seshia, A. Tiwari, Oracle-guided component-based program synthesis, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), 2010, pp. 215–224.

[6] S. Gulwani, S. Jha, A. Tiwari, R. Venkatesan, Synthesis of loop-free programs, in: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, pp. 62–73. `doi:10.1145/1993498.1993506`.

[7] S. Jha, S. A. Seshia, A theory of formal synthesis via inductive learning, CoRR abs/1505.03953.

[8] Semiconductor IP Market by Form Factor (ICs IP, SOCs IP), Design Architecture (IP cores (Hard IP, Soft IP), Standard IP, Custom IP, Processor Design), Processor Type (Microprocessor, DSP), Verification IP - Global forecast to 2022, marketsandmarkets.com (2016).

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

[10] A. Iannopollo, P. Nuzzo, S. Tripakis, A. Sangiovanni-Vincentelli, Library-based scalable refinement checking for contract-based design, in: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, 2014, pp. 1–6. doi:10.7873/DATE.2014.167.

[11] P. Nuzzo, J. Finn, A. Iannopollo, A. Sangiovanni-Vincentelli, Contract-based design of control protocols for safety-critical cyber-physical systems, in: Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, 2014, pp. 1–4. doi:10.7873/DATE.2014.072.

[12] A. Iannopollo, S. Tripakis, A. Sangiovanni-Vincentelli, Constrained synthesis from component libraries, in: O. Kouchnarenko, R. Khosravi (Eds.), Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besanccon, France, October 19-21, 2016, Revised Selected Papers, Springer International Publishing, 2017, pp. 92–110. doi:10.1007/978-3-319-57666-4_7.

[13] R. Alur, S. Moarref, U. Topcu, Compositional synthesis with parametric reactive controllers, in: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC '16, ACM, New York, NY, USA, 2016, pp. 215–224. doi:10.1145/2883817.2883842.

[14] P. J. Ramadge, W. M. Wonham, Supervisory control of a class of discrete event processes, SIAM Journal on Control and Optimization 25 (1) (1987) 206–230. arXiv:https://doi.org/10.1137/0325013, doi:10.1137/0325013.
URL https://doi.org/10.1137/0325013

[15] R. Ramesh, R. Lin, A. Iannopollo, A. Sangiovanni-Vincentelli, B. Hartmann, P. Dutta, Turning coders into makers: The promise of embedded design generation, in: Proceedings of the 1st Annual ACM Symposium on Computational Fabrication, SCF '17, ACM, New York, NY, USA, 2017, pp. 4:1–4:10. doi:10.1145/3083157.3083159.

[16] E. Y. Shapiro, Algorithmic Program DeBugging, MIT Press, Cambridge, MA, USA, 1983.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E. Emerson, A. Sistla (Eds.), Computer Aided Verification, Vol. 1855 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 154–169. doi:10.1007/10722167_15.

[18] R. Alur, S. Tripakis, Automatic synthesis of distributed protocols, SIGACT News 48 (1) (2017) 55–90. `doi:10.1145/3061640.3061652`.
URL `http://doi.acm.org/10.1145/3061640.3061652`

[19] R. Alur, M. Raghothaman, C. Stergiou, S. Tripakis, A. Udupa, Automatic completion of distributed protocols with symmetry, in: CAV (2), Vol. 9207 of Lecture Notes in Computer Science, Springer, 2015, pp. 395–412.

[20] R. Alur, M. M. K. Martin, M. Raghothaman, C. Stergiou, S. Tripakis, A. Udupa, Synthesizing finite-state protocols from scenarios and requirements, in: Haifa Verification Conference, Vol. 8855 of Lecture Notes in Computer Science, Springer, 2014, pp. 75–91.

[21] S. A. Seshia, Combining induction, deduction, and structure for verification and synthesis, Proceedings of the IEEE 103 (11) (2015) 2036–2051. `doi:10.1109/JPROC.2015.2471838`.

[22] A. Sangiovanni-Vincentelli, Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design, Proceedings of the IEEE 95 (3) (2007) 467–506.

[23] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. A. Henzinger, K. G. Larsen, Contracts for system design, Foundations and Trends in Electronic Design Automation 12 (2-3) (2018) 124–400. `doi:10.1561/1000000053`.
URL `http://dx.doi.org/10.1561/1000000053`

[24] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, C. Sofronis, Multiple viewpoint contract-based specification and design, in: F. S. Boer, M. M. Bonsangue, S. Graf, W.-P. Roever (Eds.), Formal Methods for Components and Objects, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 200–225. `doi:10.1007/978-3-540-92188-2_9`.

[25] A. Sangiovanni-Vincentelli, W. Damm, R. Passerone, Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems, European Journal of Control 18 (3) (2012) 217–238.

[26] P. Nuzzo, A. Iannopollo, S. Tripakis, A. Sangiovanni-Vincentelli, Are interface theories equivalent to contract theories?, in: Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on, 2014, pp. 104–113. `doi:10.1109/MEMCOD.2014.6961848`.

[27] L. de Alfaro, T. A. Henzinger, Interface automata, in: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, ACM, New York, NY, USA, 2001, pp. 109–120. `doi:10.1145/503209.503226`.

[28] L. De Moura, N. Bjørner, Z3: An efficient smt solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 337–340.

[29] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559, Springer-Verlag New York, Inc., New York, NY, USA, 2014, pp. 334–342. doi:10.1007/978-3-319-08867-9_22.

[30] A. Pnueli, The temporal logic of programs, in: Proceedings of the 18th Annual Symposium on Foundations of Computer Science, SFCS '77, IEEE Computer Society, Washington, DC, USA, 1977, pp. 46–57. doi:10.1109/SFCS.1977.32.

[31] A. P. Sistla, E. M. Clarke, The complexity of propositional linear temporal logics, J. ACM 32 (3) (1985) 733–749. doi:10.1145/3828.3837.

[32] D. Hanselman, Brushless permanent magnet motor design, The Writers' Collective, Cranston, R.I, USA, 2003.