Synthesizing LTL Contracts from Component Libraries Using Rich Counterexamples

Antonio Iannopollo[a], Inigo Incer[a,b], Alberto L. Sangiovanni-Vincentelli[a]

[a] *University of California, Berkeley, CA*
[b] *California Institute of Technology, Pasadena, CA*

Abstract

We provide a method to synthesize an LTL Assume/Guarantee (A/G) specification, or contract, as an interconnection of elements from a library, each of which is also represented by an LTL A/G contract. Our approach, based on counterexample-guided inductive synthesis, leverages an off-the-shelf model checker to reason about infinite-length counterexamples and guarantee correctness. To increase scalability, we also introduce a novel concept of specification decomposition, based on contract projections; we show how it can be used to break down our synthesis problem into several simpler tasks, without reducing the size of the solution space. We test our technique on three industry-relevant case studies.

Keywords: counter-example-guided inductive synthesis, assume-guarantee contracts, linear temporal logic, specification decomposition

1. Introduction

System design is a process that starts with a high-level concept of what the system should be and yields a blueprint for its construction. During design, the initial idea is refined iteratively until no ambiguity is left, proceeding in a top-down fashion. Requirements about the system as a whole are used to define specifications for its subsystems, which, in turn, are broken down into specifications for lower-level components, giving design a fractal nature [1, 2]. At each step, increasing detail about the final design is added, until there is enough information to build the system.

To reduce cost and shorten time-to-market, modern design methodologies emphasize design reuse by leveraging platforms, which are libraries of parametric components that include functional and non-functional details and define rules for their composition. A design iteration then becomes a mapping process in which a specification is realized in terms of platform components. Consider, for instance, the automotive design process. Original Equipment Manufacturers (OEMs), e.g., BMW or Ford, design several vehicles adapting the same floorplan, axles, etc., and by incorporating subsystems, such as suspension control, provided by tier 1 suppliers. In turn, tier 1 suppliers can build subsystems to order by adapting their own platform, and using components provided by tier 2 suppliers. This approach to design is formalized in a methodology called Platform-Based

Design (PBD), which has been successfully applied in areas including electronic, automotive, and network design [1, 3, 4].

When system complexity increases, especially for safety-critical systems, maintaining consistency between design iterations and level of abstraction is an onerous task. When done manually, in fact, the refinement of a specification in terms of a composition of platform elements can introduce errors, and the inevitable design review sessions are costly and ineffective [5]. Formal languages, such as temporal logic [6, 7], can help to formalize specifications and maximize the benefits of methodologies, such as PBD, by enabling automated reasoning. Specifications can be effectively described, at different levels of abstraction, by mathematical constraints which define what behaviors are to be expected by a correct implementation provided some assumptions on the environment in which the implementation will need to operate. At the same time, components in the platform expose formal descriptions of their capabilities, which are formally consistent with the specification, meaning that the verification of compliance reduces to solving a mathematical problem. The difference between specification and components is, then, quantitative. The specification will likely describe a smaller set of desirable, higher-level properties of the system. It will not include, however, all the details of the necessary subsystems, which will be added to the design only when a suitable platform component is selected and instantiated. For instance, a specification might define what is the desired voltage from a battery module to power a device. A component in the library could satisfy the voltage requirement, and it might add additional constraints such as safety measures to manage overheating, or details about its interconnections. Once verified, then, the mapping process of PBD produces an instance of the platform, where each component represents a new specification for the successive iteration. We believe this approach is necessary for the widespread adoption of formal methods in system design as it frees the designer from having to think in terms of mathematical formulas, focusing on higher-level components.

Having a formal description of design elements brings at least two substantial benefits to the design process. First, it allows for the automatic verification that a composition of platform elements implements, or refines, the specification. Second, it enables synthesis, i.e., the automatic generation of a composition of components which refines the specification. However, especially when described by temporal constraints, scalability remains an issue even for simple designs. In this paper, we describe methods to handle complexity when design elements are described by temporal constraints, and improve scalability.

1.1. Paper Overview

In this paper, we focus on specifications and platform components characterized by formal interfaces. Each design element specifies its static interface as a set of inputs and output ports and defines its set of behaviors by means of Linear Temporal Logic (LTL) formulas [6, 8]. LTL is especially useful in expressing specifications for discrete systems, where the ordering of events matters more than their precise timing. A number of domains are functionally characterized well by LTL. Examples include software, communication protocols, motion planning, reactive systems, and digital components in general [7, 9–14].

To successfully apply techniques and methodologies that support automated reasoning, these design elements need to interact with each other in a well-defined manner. Components first need a mechanism to be interconnected to

represent a single, coherent subsystem which then needs to be compared to the specification, to verify correctness. Thus, design elements need to support composition (horizontal relation), and refinement verification (vertical relation). We leverage Assume/Guarantee (A/G) contracts [15–20] to rigorously characterize components interfaces. Component ports, then, represent contract variables, and LTL formulas describe explicitly assumptions, i.e., what the component expects from its environment, and guarantees, i.e., its promise. The resulting LTL A/G contracts, analyzed in Section 3, formalize notions such as compatibility (are there valid environments for a component?) and consistency (are there valid implementations?), and support composition and refinement. Contracts, in this case, can be seen as the language unifying PBD, where all the considerations made about the libraries of components (platforms) apply to libraries of contracts, too. Hence, both the system specification and the platform components are described by LTL A/G contracts, and contract libraries represent domain knowledge that will be added to the design upon the instantiation of their elements. Our goal is to automate, for such contracts, the mapping process of PBD. The result will be a formally correct instantiation of the platform, which in this case will be a composition of LTL A/G contracts. Therefore, in this paper, we talk about synthesis of LTL specifications.

We focus our attention on the synthesis of correct compositions of contracts. In [21], we addressed this problem of synthesis assuming that the only output of the refinement verification procedure is a simple yes/no answer. In that paper, we devised a synthesis procedure based on the Oracle-Guided Inductive Synthesis (OGIS) paradigm [22, 23], where erroneous candidate solutions are used to infer new constraints to guide the synthesis process. In that case, the only information available to the solver is the erroneous candidates themselves, which are used to identify equivalent compositions in the library so that they won't appear as candidate solutions in the future. The main advantage of this approach is that the verification tasks are independent of one another, allowing for the parallel execution of the resulting algorithm. Additionally, as we already point out in [21], this technique is loosely related to the use of LTL A/G contracts, and we argue that it could be applicable to other formalisms, too.

In this paper, we change our perspective and focus exclusively on LTL A/G contracts. We tighten the assumptions on the refinement verification process, that now is required to return, when refinement does not hold, counterexample traces over the contract variables in addition to the usual yes/no answer. Hence, here we talk about *rich* counterexamples as each iteration provides information on both a candidate topology and its behavior. The result is a procedure based on a specialization of OGIS called Counterexample-Guided Inductive Synthesis (CEGIS) [22, 24, 25] where we deal with infinite-length counterexamples by encoding them as state machines. Each state machine, then, is integrated into the refinement verification process of the subsequent CEGIS iteration. For this problem, we show that our approach indeed terminates in spite of infinite input space, and discuss several performance improvement techniques.

We also study the problem of scalability of the synthesis techniques we developed. We do so by leveraging contract properties to decompose, under certain conditions, a specification in several independent synthesis problems. Each synthesis task is simpler than the original one, as the resulting decomposed specification will have fewer ports to be mapped into a candidate composition of components. Thus, this allows us to handle synthesis problems that are

unfeasible with the other techniques. We apply all the synthesis strategies that we introduce to several case studies. The software and experiments discussed in this paper have been implemented as an extension of the tool PYCO[1].

1.2. Main Contributions and outline

This paper extends our previous work in [21, 26]. The main contributions are the following: we provide a synthesis approach of LTL A/G contracts based the CEGIS paradigm (Section 4), substantially different than the approach in [21]. A novel concept of contract decomposition, revised from [26], is introduced in Section 5, based on projections; we show how it can be used to break down our synthesis problem into several simpler tasks, without reducing the size of the solution space.

The paper is organized as follows: in Section 2, we survey the related literature and compare it with our work. We formalize the core concepts that will support the rest of the work in Section 3. There, we introduce A/G contracts and describe a variant of the theoretical framework in which contracts can be defined over disjoint sets of variables and connections are explicitly referenced. Additionally, we show how to express assumptions and guarantees of contracts as LTL formulas. In Section 4, we define the synthesis problem for LTL A/G contracts and its solution, where the verification engine also returns counterexamples over the contract variables. The solution is a CEGIS algorithm that encodes those counterexamples as state machines. In Section 5, we discuss how to improve the scalability of the synthesis algorithms described in the previous section by decomposing specifications, introducing the notion of projection for LTL A/G contracts. We evaluate the proposed techniques on three case studies, discussed in Section 6. In Section 7, we draw conclusions and discuss future research directions. Finally, Appendix A includes additional details on techniques and tools to perform basic operations on LTL A/G contracts, such as refinement verification.

2. Related Works

Contract Refinement Verification. The problem of building and verifying compositions of formally defined components has been studied extensively. Pinto *et al.* [27] propose the contract-based Specification Language for Platforms (CSL4P), which provides mechanisms to define component platforms and to build designs by instantiating, interconnecting, and composing components. Designs can then be formally verified for compliance with platform composition rules. Grumberg and Long [28] describe the idea of decomposing a verification task into smaller sub-tasks, where an aggregation of components is replaced by a more abstract representation, according to an Assume/Guarantee framework. However, in most cases, finding the appropriate abstraction is an issue since no general guidelines are available to the verification engineer. A few approaches have been proposed, which use learning algorithms to automatically build such abstractions, e.g., see [29, 30]. In [31], the abstraction process is instead guided by the contract library, which systematically encodes the available information on both the structural decomposition of the system architecture and the relevant system domain

[1] `https://github.com/ianno/pyco`

knowledge. Based on the library, we provide a mechanism to automatically build abstractions on the fly, as we solve the problem by successive refinements. In this respect, our solution was inspired by the PBD paradigm [1] where a design, at each abstraction layer, is also regarded as a platform instance, i.e., a valid interconnection of component out of a pre-characterized library, which also includes composition rules. As we describe in Section 3.3, the concept of library is further extended to also include relations between contracts and their ports. In the context of [31], such relations include refinement rules between contracts in the library. Cimatti and Tonetta [32] exploit the relation between decomposition of component contracts and system architecture and provide a concrete framework to verify a system architecture relying on temporal logic formulas.

Synthesis. In [33], the problem of synthesizing a fixed network topology of finite-state machines that cooperate to satisfy a given LTL specification is shown to be undecidable. In [34], the problem of synthesizing a network topology of given components that cooperate to satisfy a given LTL specification is also shown to be undecidable. In our previous work [21] and in this paper, we evade these undecidability results by imposing a bound on the number of components in the network topology. In [23, 35], the authors consider the problem of synthesis from component libraries, but their components are finite loop-free programs, as opposed to LTL formulas, which we consider. Similarly, [36] considers the synthesis of embedded designs from component libraries. In this work, components satisfy static specifications, while we consider components satisfying temporal constraints. In [37], the authors consider component-based synthesis of a system in which each component takes full control of the entire system when it is active. In contrast, we consider networks of components that must collaborate in order to satisfy a top-level specification.

We previously addressed the problem of synthesis from component libraries in [21]. In that paper, we focused on the problem of synthesis from libraries of components when minimal assumptions were made on both the formalism used to describe components and the verifier, i.e., an oracle able to determine the correctness of a certain candidate solution. The only requirements imposed on the chosen formalism were being able to compose components and, given two different components, being able to determine if one is the refinement of the other. Furthermore, we only required the verifier to provide a yes/no answer. That synthesis approach relies on generating many candidate solutions, each of which can be independently verified. Specifically, the size of each verification problem depends only on the specification itself and the candidate solution, not on the number of tested candidates. The methods presented in this paper, instead, focus solely on LTL A/G contracts and use a model checker as a verifier, leveraging its ability to return counterexample traces. This choice allows us to harness properties unique to LTL and A/G contracts, enabling us to go beyond just observing the Boolean outcome of the refinement checking process. Indeed, we can access more detailed information provided by the underlying model checker, i.e., counterexamples representing complete system runs, and use them to improve the efficiency of the synthesis process. We propose an approach based on CEGIS that utilizes the infinite-length counterexamples generated by the model checker, which we compactly represent as state machines, to guide the synthesis process. This synthesis strategy, in comparison to the one used in

[21], is more efficient as it requires fewer calls to the verifier, although generating candidate solutions is more work-intensive. We observe, therefore, that our synthesis strategy does not rely on existing LTL synthesis software, but on LTL model-checking, leveraging mature, well-known tools [38] that we use as a black box.

Oracle-Guided and Counterexample-Guided Inductive Synthesis, and the combination of Induction, Deduction, and Structure. OGIS [22, 23] is a general paradigm to address formal inductive synthesis. It is characterized by the interaction between an inductive learning engine, also called the "learner", and a verification engine, i.e., the "teacher". The learner submits queries to the teacher, which replies with some information (for instance, a yes/no answer or an execution trace) that is used by the learner to improve its guesses.

CEGIS [22, 24, 25], a specialization of OGIS, is a well-known synthesis paradigm that originates from techniques of debugging using counterexamples [39] and *counterexample-guided abstraction refinement* (CEGAR) [40]. CEGIS is an inductive synthesis approach where synthesis is the result of inferring details of the solution from I/O examples, which usually are counterexamples for previous incorrect guesses, provided by a constraint solver. In CEGIS an iterative algorithm, according to a certain concept class, generates candidate solutions that are processed by an oracle and either declared valid, in which case the algorithm terminates, or is used to derive counterexamples to restrict the search space. CEGIS has been successfully used in a number of research areas, including program synthesis and sketching [24, 35], and synthesis and completion of distributed protocols [41–43]. Seshia [44] proposed a methodology that formalizes the combination of *Structure, Inductive and Deductive* reasoning (SID), representing a generalization of both CEGAR and OGIS.

Specification Decomposition. Filippidis [20] studies the problem of specification decomposition into A/G contracts, when they are described in a fragment of Temporal Logic of Actions (TLA$^+$) [45], by proving that unnecessary variables can be efficiently hidden and eliminated from the resulting specifications. Henzinger *et al.* [46] propose a method to decompose the refinement verification process for reactive systems in a series of sub-tasks that are simpler than the original problem, leveraging the structure of the design and using the Assume/Guarantee paradigm to manage circular dependencies. Our goal is similar: breaking down the synthesis process into simpler sub-tasks. Dallal and Tabuada [47], given a set of components and a safety specification, propose a decomposition technique where the goal is to generate a set of minimally restrictive assumptions (one per component). Such assumptions, found through a fixed point computation, are then used to synthesize controllers for the components. Incer *et al.* [48] introduce means to compute the operation of quotient for A/G contracts. Given a specification C to be implemented, and the specification C' of a component to be used in the design, the quotient describes the properties that need to be satisfied, in addition to those required by C', in order to meet C.

3. Preliminaries

In this section, we provide some basic notions that will be the foundation of the work developed in the next sections. Our goal is two-fold. On the one

hand, we describe the formalisms that we use later in the paper and how they relate to each other. For instance, we introduce Linear Temporal Logic (LTL), and discuss how it can be used to specify A/G contracts. On the other hand, we want to provide the reader with an understanding of the more general A/G contract framework, thus expanding some concepts and referring, as much as possible, to definitions and notions from the current literature. We invite readers familiar with these topics to skip this material and proceed to the contributions of this paper, starting in Section 4.

3.1. Linear Temporal Logic

Temporal Logic is an extension of propositional logic introduced by Amir Pnueli [6] that allows for the specification of properties that can be verified over an infinite sequence of symbols. Here we focus on Linear Temporal Logic (LTL), which is particularly useful in expressing properties of systems having a state that evolves in a discrete manner, where time is seen as a linear sequence in which system variables are evaluated. Programs, electric and electronic devices, and motion planning are just a few examples of areas that can benefit from having specifications expressed using LTL. LTL is expressive enough to describe complex specifications, including properties such as safety (something bad will never happen), liveness (something will keep happening), stability (a certain state will be eventually reached), etc. [7]

3.1.1. Reactions, Behaviors, and Synchronous Assertions

Given a set of variables Σ with domain D, we call *reaction*, or *state*, $r \in D^\Sigma$ an evaluation of the variables in Σ within their domain. A synchronous run σ, or *behavior*, is an infinite sequence of reactions:

$$\sigma \in (D^\Sigma)^\omega = r_0, r_1, r_2, \cdots \tag{1}$$

A set of behaviors is called a synchronous assertion. A synchronous assertion P is defined as:

$$P \subseteq \mathcal{T} \tag{2}$$

where $\mathcal{T} = (D^\Sigma)^\omega$ is the set of all the behaviors.

We call *atomic proposition* any statement over evaluations of variables in Σ which has a unique truth value. This means that for an atomic proposition p is always possible to determine if it is true or false. Given a reaction $r \in D^\Sigma$, we say that $r \vdash p$ if p is true over the evaluation in r. If p is false, then $r \nvdash p$.

3.1.2. Syntax of LTL formulas

Given a set of atomic propositions AP over evaluations of an alphabet Σ, the syntax of an LTL formula φ can be defined inductively as follows:

$$\varphi := \mathit{True} \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \psi$$

where $p \in AP$ and ψ is another LTL formula, $\bigcirc$ is the *next* operator—also indicated as X—and $\mathcal{U}$ is the *until* operator.

Additional logic and linear temporal operators, such as disjunction ($\vee$), material implication ($\rightarrow$), eventually ($\Diamond$, or F), and globally ($\Box$, or G) can be

derived as follows:

$$\begin{aligned} \varphi \vee \psi &:= \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \rightarrow \psi &:= \neg\varphi \vee \psi \\ \Diamond\varphi &:= \mathit{True}\ \mathcal{U}\ \varphi \\ \Box\varphi &:= \neg\Diamond\neg\varphi \end{aligned}$$

3.1.3. Semantics of LTL formulas

LTL formulas are evaluated over infinite sequences of states, i.e., behaviors. Let $\sigma = r_0, r_1, \cdots$ be a behavior. Then, the satisfaction of a formula φ by σ at time i is recursively defined as follows:

- $\sigma_i \models p$ if and only if $r_i \vdash p$, where $p \in AP$;
- $\sigma_i \models \neg\varphi$ if and only if $\sigma_i \not\models \varphi$;
- $\sigma_i \models \varphi \wedge \psi$ if and only if $\sigma_i \models \varphi$ and $\sigma_i \models \psi$;
- $\sigma_i \models \bigcirc\varphi$ if and only if $\sigma_{i+1} \models \varphi$;
- $\sigma_i \models \varphi\,\mathcal{U}\,\psi$ if and only if there exist $j \geq i$ such that $\sigma_j \models \psi$ and $\sigma_k \models \varphi$ for all $k \in [i, j)$;

where ψ is another LTL formula. If a formula is true at time 0, i.e., $\sigma_0 \models \varphi$, we will drop the subscript and just say that the behavior σ satisfies the formula φ, written $\sigma \models \varphi$.

For an LTL formula φ, we indicate with $\mathcal{L}(\varphi)$ its language, i.e., its associated synchronous assertion, which is the set of infinite behaviors satisfying it.

3.1.4. LTL Applications, satisfiability, and realizability

LTL is widely used in model checking and synthesis [12, 28, 38, 49, 50]. In the first case, it is possible to automatically verify if a model exhibits certain properties. Several model-checking algorithms and tools have been developed over the years. These algorithms can be grouped into two categories: symbolic and explicit-state; symbolic algorithms use Binary Decision Diagrams (BDD) [51] to encode the whole state space and perform model checking. Explicit state algorithms [52], instead, declare all the state variables for each time step and then rely on a SAT solver to find an answer to the model-checking problem.

In the case of synthesis, given a set of desirable LTL properties, the goal is to automatically generate a state machine implementing them. This is achieved by solving a two-player game, where a system implementation is required to be able to react to any possible move from an adversarial environment [49, 50, 53]. If a winning strategy does not exist, then synthesis fails. Otherwise, an implementation of such a strategy is returned.

In this work, however, we verify and synthesize compositions of components described using LTL, and all the algorithms and techniques we study are based on the capability of asserting the validity of an LTL formula. In the previous sections, we discussed how LTL formulas can express sets of infinite-length behaviors. Indeed, Wolper *et al.* [54] show that the models satisfying an LTL formula can be described as an ω-regular language over a certain alphabet. Thus,

checking the validity of an LTL formula can be reduced to checking emptiness of the associated language, which is a PSPACE-complete problem [8].

There are several options to solve the LTL validity problem for a formula φ. One of them is to use a tool that is able to compute an automaton that accepts infinite-length words, called *Büchi* automaton, corresponding to the ω-regular language accepted by φ. If the automaton is empty, then the formula is not satisfiable, i.e., its negation is valid. For instance, a tool able to compute such automaton is LTL2BA [55].

Another method is to use a model checker to check whether an unconstrained model is able to satisfy the property φ. Model checkers perform similar automata-based reasoning over the formula and the language generated by the system, but they are often faster due to the efficient symbolic encoding that many of them use. Some other model checkers use BMC to verify the validity (up to a certain temporal horizon) of a formula φ. In our experience, if having a finite temporal horizon is acceptable, this is the solution that yields the best performance for large formulas. Using a model checker to check for LTL validity has also the advantage of generating counterexamples when a certain formula φ is not valid. Those counterexamples can then be used to infer a satisfiable assignment for the negated $\neg\varphi$.

The realizability of an LTL formula is also seen as a game played by two players. It was studied initially by Pnueli and Rosner in the context of LTL synthesis [49]. For an LTL formula φ, each player controls a subset of the formula propositions by controlling a subset of its variables. Thus, the environment controls a set I of input signals, while the system to be synthesized controls the set O of output signals. The goal of the system is to satisfy φ, while the environment wants to falsify it, without falsifying its subset of propositions. The game is played in turns, where for each variable value assignment from the environment there is a corresponding assignment from the system. The result is an infinite sequence of reactions, i.e., a behavior. If the behavior satisfies φ, then the system wins, otherwise the environment wins. The formula is realizable if the environment never wins.

LTL realizability is a 2EXPTIME-complete problem [50], but there are several tools that are able to compute it by implementing different strategies. For instance, we mention the design tools RATSY [56] and Tulip [57], or the LTL synthesis tools ACACIA+ [53, 58]. Later, we will show how realizability can be used to prove some properties on LTL A/G contracts, such as consistency and compatibility.

3.2. Design Contracts

The concept of design contracts, which has been extensively studied in the past few years [15–17, 19, 20, 59, 60], has its roots in the field of software engineering, where assume/guarantee reasoning has traditionally been used to reason about pre- and post-conditions of software modules [61]. This approach to software engineering has been derived from, in turn, the early work of Floyd and Hoare [62, 63]. The shift of the use of contracts towards system design, however, has been influenced by the early work on interface theories [18, 64–66].

The concept of contract nicely embraces the discipline of system design, as it emphasizes modularity and reuse of components, which are critical elements of the practices followed in industry. Indeed, when developing a system, several

suppliers collaborate with an Original Equipment Manufacturer (OEM) on the basis of some partial specifications. Such specifications require a supplier to develop a component that is able to guarantee a certain functionality, assuming a certain set of constraints on its operative environment. If designed according to the specification, each component will be able to properly interact with other components, even when they have been realized by different suppliers.

The modern theory of design contracts is broad and encapsulates many concrete theories developed over the years for concrete applications. In this paper, we focus on assume/guarantee contracts, where the set of acceptable system and environment behaviors are explicitly formalized. For a full analysis and description of the theory of contract for system design, we refer to [16, 60].

In our work, the description of a design unit, or just a component in the remainder of this section, follows Benveniste's definition in [16] that is, in turn, inspired by the Tagged Signal Model developed by Lee and Sangiovanni-Vincentelli [67]. A component implementation $M = (\Sigma, P)$ is a specific realization of a component. It refers to a certain set Σ of ports, or variables, with domain D. It is characterized by an assertion P, i.e., a set of behaviors that represent a valid execution, or run, of the component

Given a certain design goal, several components can achieve it by properly working together. Such collaboration is realized by composing components according to some well-defined rules. Two implementations M_1, M_2 over the same set of variables Σ can interact with each other by composing them. In this case, composition means that there exists a non-empty set

$$P_{\|} = \{\sigma \mid \sigma \in P_1 \text{ and } \sigma \in P_2\} = P_1 \cap P_2 \tag{3}$$

That is, M_1 and M_2 agree on some possible executions, and the composed implementation is indicated as

$$M_{\|} = M_1 \parallel M_2 = (\Sigma_P, P_1 \cap P_2) = (\Sigma_P, P_{\|}) \tag{4}$$

In the following sections, we will also use the term component to indicate a generic element in a set or library. To avoid ambiguity, outside the context of this section, we will use the term *design unit* to indicate a component as those defined in this section.

3.2.1. Assume/Guarantee Contracts

An Assume/Guarantee (A/G) contract is a description of a component which decouples the responsibilities of the component itself, i.e., its guarantee, from the responsibilities it assumes on its environment. A/G contracts are also defined using synchronous assertions. In this paper, we specialize the description of A/G contracts from [15, 17] to explicitly handle input and output variables.

Definition 3.1. *An A/G contract is a tuple* $C = (I, O, A, G)$[2] *where* $I \subseteq \Sigma$ *is a set of input variables,* $O \subseteq \Sigma$ *is a set of output variables, and* Σ *is the contract*

[2]This is the definition of contracts found in [17]. Damm et al. [68] define a contract as an object (A, B, G), where A and B are strong and weak assumptions, respectively, and G contains the guarantees. Strong assumptions are used to describe conditions that are needed for a component to operate. Weak assumptions describe environments in which the component provides reduced functionality.

alphabet, which is assumed to be the same for all contracts. A and G, instead, are synchronous assertions representing assumptions and guarantees, respectively.

The pair $\pi = (I, O)$ is called a profile, and represents the partition of variables that can and cannot be controlled by the contract[3]. Having such a clear partition is extremely useful, as it allows us to clearly identify which variables can be used by a contract to carry out its promise. An assertion P is called Σ'-receptive, where $\Sigma' \subseteq \Sigma$ is a set of variables, if and only if for all behaviors σ' defined over variables in Σ', there exists a behavior $\sigma \in P$ such that $\prod_{\Sigma'}(\sigma) = \sigma'$, where $\prod$ indicates the projection operation, i.e., when σ is considered only over variables in Σ'. Thus, P accepts any possible sequence over variables in Σ'. Sometimes, when the context is clear, we refer to a contract omitting its profile. For instance, we could refer to the contract $C = (I, O, A, G)$ simply as $C = (A, G)$.

Consider a contract $C = (I, O, A, G)$. Any assertion $E \subseteq A$ is a valid environment for C, indicated as $E \models_{\mathscr{E}} C$, while any assertion M such that $M \cap A \subseteq G$, indicated as $M \models_{\mathscr{M}} C$, is a valid implementation, i.e., the component is behaving correctly under the assumptions of the contract.

Different implementations can, in general, satisfy the same contract. Consider, for instance, two implementations M_1, M_2 such that $M_1 \neq M_2$, and a contract $C = (I, O, A, G)$ We can still have both $M_1 \cap A \subseteq G$ and $M_2 \cap A \subseteq G$. We refer to the maximal implementation for C as $M_C = G \cup \overline{A}$, where $\overline{A} = \mathcal{T} \setminus A$ indicates the complement of A. For any implementation M such that $M \models_{\mathscr{M}} C$, we have that $M \subseteq M_C$.

Saturated Contracts. Consider, two contracts $C_1 = (A, G_1), C_2 = (A, G_2)$. Let C_1, C_2 have different guarantees, i.e., $G_1 \neq G_2$, but identical maximal implementations $M_{C_1} = M_{C_2}$. Thus, we have

$$M_{C_1} \cap A = (G_1 \cup \overline{A}) \cap A = (G_2 \cup \overline{A}) \cap A \tag{5}$$

This implies that the difference between the guarantees is not included in the assumption A,

$$G_1 \Delta G_2 \subseteq \overline{A} \tag{6}$$

where Δ is the symmetric difference between two sets. If it were, then Equation 5 would not hold, contradicting our assumption that C_1 and C_2 share the same maximal implementation. Consider now the contract $C = (A, G)$, whose maximal implementation is $M_C = M_{C_1} = M_{C_2}$. Let also $G \supseteq \overline{A}$. Thus, it follows from Equation 6 that G is maximal, meaning that, for any contract $C' = (A, G')$ with maximal implementation M_C, we have $G' \subseteq G$. In this case, we say that the contract C is *saturated*, meaning that it explicitly contains the largest possible guarantees for a certain maximal implementation. For a saturated contract C, we also have that $G \cup A = \mathcal{T}$, meaning that the union of assumption and guarantee includes all the possible behaviors. Saturated contracts are useful because they remove ambiguities between contracts that have the same sets of satisfying implementations. Unless differently indicated, we will always refer to saturated contracts.

[3]In [15, 17], the terms *input* and *output* are replaced by the terms *uncontrolled* and *controlled*, respectively, to stress the extent of assumptions and guarantees over a contract's variables.

Compatibility and Consistency. Sometimes, an A/G contract is ill-defined, meaning that it specifies a contradictory assumption or guarantee. If there are no suitable implementations that can conform to the contract, we say that the contract is *inconsistent*. Formally, we say that a contract $C = (I, O, A, G)$ is inconsistent if G is not I-receptive. This means that there are some behaviors consistent with the contract assumption, that will falsify the guarantee G, and that cannot be avoided by any implementation of C. The only way for the contract to be satisfied, is for an implementation to impose constraints over input variables, which is a contradiction. Conversely, a contract that is I-receptive is called consistent.

On the other hand, if there is no suitable environment for a contract, we say that the contract is *incompatible*. A contract $C = (I, O, A, G)$ is incompatible if and only if A is not O-receptive. In opposition to the consistency case, an incompatible contract could generate a sequence of evaluations over its controlled variables which is rejected by every possible environment. Thus, it would be impossible for an environment to fully comply with the contract assumption without controlling some contract variables, which is against the principle of separation of concerns between a system and its environment. An O-receptive contract is, instead, compatible.

Definition 3.2. *C is well-defined if and only if it is consistent and compatible, $I \cap O = \emptyset$, and A and G are defined over variables in $I \cup O$.*

If not differently mentioned, we will always consider well-defined contracts.

Parallel Composition. Often complex components are realized by having simpler components working together. In the same way, complex contracts can be built by composing simpler ones. Formally, we can compose contracts with each other through parallel composition. The operation of parallel composition is a function that takes two contracts as input and returns a third one, i.e., their composition:

$$\|: \mathbb{C} \times \mathbb{C} \to \mathbb{C}$$

where $\mathbb{C}$ is the set of all A/G contracts.

Specifically, given contracts $C_1 = (I_1, O_1, A_1, G_1), C_2 = (I_2, O_2, A_2, G_2)$, their composition $C = (I, O, A, G) = \| (C_1, C_2)$—also expressed using the infix notation $C = C_1 \parallel C_2$—is defined as follows:

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \tag{7a}$$

$$O = O_1 \cup O_2 \tag{7b}$$

$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \tag{7c}$$

$$G = G_1 \cap G_2 \tag{7d}$$

In principle, a contract representing the composition of two simpler ones should guarantee behaviors that are consistent with both its constituent contracts. Thus, it seems natural that the guarantee of the composed contract is formulated as the intersection of the guarantees of those contracts. For the assumption, however, intersecting the assumptions might be too restrictive, as sometimes a contract provides some inputs of another contract that it is being composed with. The composition, indeed, should consider that some requirements on the environment assumed by a constituent contract could be already fulfilled in the

composition itself. The definition above reflects this intuition. The environment of the newly composed contract is relieved of the responsibility of exposing those behaviors which are already guaranteed by the constituent contracts. Thus, if any of the constituent contracts does not keep its promise, the assumption on the environment is trivially satisfied.

Similarly, we can intuitively justify the definition of the profile of the composition in Equations 7a and 7b. The set of output ports of the contract is simply the union of the output ports of the contracts being composed. The set of input ports, however, needs to consider that some input ports of a constituent contract might be controlled by another contract, thus are no longer the responsibility of the environment.

Parallel composition preserves saturation, meaning that C_1 and C_2 are saturated, so is their composition, but it does not necessarily preserve consistency or compatibility. For instance, it is always possible to compose two contracts guaranteeing contradicting assertions. While they might be individually consistent, the intersection of their guarantees will be empty, meaning that no implementation can satisfy the composition.

Parallel composition is commutative and associative, as shown in Chapters 4 and 5 of [16]. This means we can generalize composition to an arbitrary number of contracts while disregarding the order of operations. For instance, for the composition C of contracts C_1 to C_3, we will simply write $C = C_1 \parallel C_2 \parallel C_3$.

Refinement. The refinement relation between contracts is the formalization of a notion of substitutability between them. Informally, a contract can be used *in lieu* of another one if it accepts a larger set of environments and guarantees a subset of the original contract's behaviors. Thus, we say that a contract $C' = (I', O, A', G')$ *refines* a contract $C = (I, O, A, G)$, written as $C' \preceq C$, if and only if

$$I' \subseteq I \tag{8a}$$
$$O' \supseteq O \tag{8b}$$
$$A' \supseteq A \tag{8c}$$
$$G' \subseteq G \tag{8d}$$

The refinement relation is a partial order over the set of all contracts, as it is reflexive, transitive, and antisymmetric.

This notion of refinement supports independent implementability, i.e., if $C_1 \preceq C_1'$ and $C_2 \preceq C_2'$, then $C_1 \parallel C_2 \preceq C_1' \parallel C_2'$.

Contract Obligations. For a contract $C = (A, G)$, we say that its contract obligation B_C is the assertion

$$B_C = A \cap G \tag{9}$$

Intuitively, a contract obligation describes those behaviors that the contract allows in an ideal scenario, thus ignoring *bad* environments, where the contract would be trivially satisfied. For contracts C_1, C_2, we say that C_1 *conforms* to C_2 if and only if its contract obligation is included in C_2's, i.e., $B_{C_1} \subseteq B_{C_2}$. Conformance is compositional with respect to parallel composition. Consider,

for instance, contracts C_1, C_1', C_2 where C_1' conforms to C_1. Let $C_\parallel = C_1 \parallel C_2$ and $C_\parallel' = C_1' \parallel C_2$. Then, $C_\parallel'$ conforms to $C_\parallel$, as we have:

$$\begin{aligned}(A_1' \cap A_2 \cup \overline{G_1' \cap G_2}) \cap (G_1' \cap G_2) &\subseteq (A_1 \cap A_2 \cup \overline{G_1 \cap G_2}) \cap (G_1 \cap G_2)\\ (A_1' \cap A_2) \cap (G_1' \cap G_2) &\subseteq (A_1 \cap A_2) \cap (G_1 \cap G_2)\\ (A_1' \cap G_1') \cap (A_2 \cap G_2) &\subseteq (A_1 \cap G_1) \cap (A_2 \cap G_2)\\ B_{C_1'} \cap B_{C_2} &\subseteq B_{C_1} \cap B_{C_2}\\ B_{C_\parallel'} &\subseteq B_{C_\parallel}\end{aligned}$$

Conformance, however, does not imply refinement, and *vice versa*.

Contract Connection. We say that two contracts are connected if at least one input of a contract is provided by the other, i.e., for contracts C_1, C_2, we have $(O_1 \cap I_2) \cup (O_2 \cap I_1) \neq \emptyset$. Derived from the definition by de Alfaro and Henzinger in [18], we say that an *interconnect*, or renaming, θ is a set of pairs (x, y) of variables, called target and source, respectively, such that for all pairs $(x_1, y_1), (x_2, y_2) \in \theta$ we have that $x_1 \neq x_2$. Hence, θ is a partial function. For each θ, we consider an associated total function $\bar{\theta}$ defined, for a variable x, as follows.

$$\bar{\theta}(x) = \begin{cases} y & \text{if } (x, y) \in \theta \\ x & \text{otherwise} \end{cases} \tag{10}$$

A connection $\vartheta : \mathbb{C} \times \Theta \to \mathbb{C}$, where Θ is the set of all interconnects, is a function that maps a contract and an interconnect to a new contract. Given a contract $C = (I, O, A, G)$ and an interconnect θ, we indicate their connection as a new contract $\vartheta(C, \theta)$, also expressed as $C\theta$ for simplicity. We have that $C\theta = (I_\theta, O_\theta, A_\theta, G_\theta)$ is a contract where

$$I_\theta = (I \cup \{\ y \mid \exists x \in I \colon (x, y) \in \theta\ \}) \setminus O_\theta \tag{11a}$$

$$O_\theta = O \cup \{\ x \mid (\exists y \colon (x, y) \in \theta) \ \wedge\ x \in I\ \} \cup \{\ y \mid \exists x \in O \colon (x, y) \in \theta\ \} \tag{11b}$$

$$A_\theta = A \cup \overline{\rho_\theta} \tag{11c}$$

$$G_\theta = G \cap \rho_\theta \tag{11d}$$

and

$$\rho_\theta = \bigcap_{x, y \in I_\theta \cup O_\theta, (x, y) \in \theta} \{\sigma \mid x \equiv y\} \tag{12}$$

represents the set of all the behaviors where all the pairs in θ which are also referring to variables in $I_\theta \cup O_\theta$ are equivalent.

By mapping an interconnect θ to a contract C, the resulting contract $C\theta$ could have new input ports, although it will never have more inputs than the original contract. It is possible, however, for it to have more outputs than the original. This makes sense because the new outputs will represent input variables that are now controlled by some other variable, or new outputs that are *mapped* to current outputs. Assumption A_θ and guarantee G_θ also change to reflect the new relations between variables. On one side, we have that the assumption now is weaker, meaning that the contract now assumes that the variables in θ will

be equivalent, and it will consider its assumptions violated if that is not the case. The guarantee, on the contrary, is stronger, meaning that the contract is also responsible to provide some of those equivalences. Equivalently, applying a connection θ to a contract $C = (I, O, A, G)$ can be seen as the composition $C\theta = C \parallel C_\theta$, where $C_\theta = (I_\theta, O_\theta, \mathcal{T}, \rho_\theta)$, with $\mathcal{T}$ representing all the behaviors.

Note that even if contract C is well-defined, a connection operation it might render it ill-defined, i.e., inconsistent or incompatible. For instance, consider the following example.

Example 3.1 (Connection yields an inconsistent contract)**.** *Let $C = (\{a\}, , \{b\}, True, b = \neg a)$ be a contract that guarantees that its output is the negation of its input, where a and b are Boolean variables. Clearly, this contract is well-defined. Let $\theta = \{(a, b)\}$ be an interconnect that specifies a feedback loop between a and b. Thus, the connected contract is $C\theta = (\emptyset, \{a, b\}, True, (b = \neg a) \wedge (a = b))$, which is inconsistent as its guarantee cannot be satisfied.*

The following definitions use the concept of connection to introduce new relations between contracts, which will be useful in later sections.

Definition 3.3 (Contract Equivalence)**.** *Two contracts $C = (I, O, A, G)$ and $C' = (I', O', A', G')$ are said equivalent, indicated as $C \equiv C'$, if and only if there exists an interconnect θ such that the two contracts refine each other, meaning that $C'\theta \preceq C$, and $C\theta \preceq C'$.*

Definition 3.4 (Contract Copy)**.** *Given a contract $C = (I, O, A, G)$, we say that a contract $C' = (I', O', A', G') = c(C)$ is a copy of C if and only if they are equivalent, for a certain θ, and if $(I \cup O) \cap (I' \cup O') = \emptyset$. For each variable $x \in I \cup O$, we write $C'.x$ to indicate the fresh variable x' that replaced x in $I \cup O$.*

3.3. Contract Libraries

A contract library is a collection of contracts, which embeds some domain-specific knowledge. Formally, a library

$$L = (\mathcal{Z}, \mathcal{R})$$

is a pair where $\mathcal{Z} = \{C_1, C_2, \cdots, C_n\}$ is a set containing contracts defined over a common alphabet Σ and $\mathcal{R}$ defines the set of valid connections for contracts in $\mathcal{Z}$. Each contract in $\mathcal{Z}$ has unique variable names, meaning that for each pair of contracts $C_1, C_2 \in \mathcal{Z}$, we have $(I_1 \cup O_1) \cap (I_2 \cup O_2) = \emptyset$. $\mathcal{R}$ embeds library-specific rules on what connections and compositions are valid. At this point, we do not impose any specific format for the constraints in $\mathcal{R}$. We will provide a better description for them once we discuss specific libraries in the following sections. Ideally, for any interconnect θ that is applied to contracts in L, we would like $\theta \Rightarrow \mathcal{R}$, meaning that the interconnect is not in contradiction with $\mathcal{R}$. In general, one can assume $\mathcal{R}$ implying a set of pairs $(x, y) \in \Sigma^2$ of variables such that the connection of contracts in $\mathcal{Z}$ according to a certain interconnect θ is considered invalid for the library if $\theta \not\subseteq \mathcal{R}$.

3.4. LTL A/G Contracts

In Section 3.1.4, we discussed the connection between LTL formulas and sets of infinite-length behaviors, i.e., ω-regular languages. LTL formulas can be used

to represent synchronous assertions, according to our definition in Section 3.1.1. Indeed, for a formula φ, its language $\mathcal{L}(\varphi)$ indicates the related assertion.

Therefore, we can use LTL formulas to express a whole class of A/G contracts, where assumptions and guarantees are concretely expressed using a pair of LTL formulas. We call this class of contracts LTL A/G Contracts.

An LTL A/G contract is, then, a tuple $C = (I, O, \varphi, \psi)$ where φ and ψ are LTL formulas over symbols in $I \cup O$. As they are a subclass of A/G contracts, all the considerations discussed in Section 3.2.1 apply also for LTL A/G contracts, where the set operations we used—conjunction ($\cap$), disjunction ($\cup$), complement ($\overline{\ }$)—can be directly translated to formulas using logic conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), respectively.

Thus, a contract $C = (I, O, \varphi, \psi)$ is saturated if and only if $\psi \leftrightarrow (\psi \vee \neg\varphi) \Rightarrow \psi \rightarrow \varphi$, where $\rightarrow$ is the symbol for material implication and $\leftrightarrow$ is double material implication.

The composition of two contracts $C_{\|} = C_1 \parallel C_2$ is computed as:

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \tag{13a}$$
$$O = O_1 \cup O_2 \tag{13b}$$
$$\varphi_{\|} = (\psi_1 \wedge \psi_2) \rightarrow (\varphi_1 \wedge \varphi_2) \tag{13c}$$
$$\psi_{\|} = \psi_1 \wedge \psi_2 \tag{13d}$$

The verification of refinement between two contracts is also quite similar to the general case. Indeed, we say that $C' \preceq C$ if and only if:

$$I' \subseteq I \tag{14a}$$
$$O' \supseteq O \tag{14b}$$
$$\varphi \rightarrow \varphi' \text{ is valid} \tag{14c}$$
$$\psi' \rightarrow \psi \text{ is valid} \tag{14d}$$

Refinement can be efficiently verified, for LTL A/G contracts, using any tool able to check the satisfiability of LTL formulas, such as a model checker, as discussed in Section 3.1.4 and Appendix A.

Finally, for an interconnect θ and an LTL A/G contract C, we indicate the connected contract as $C\theta = (I_\theta, O_\theta, \varphi_\theta, \psi_\theta)$, where I_θ and O_θ are the same as Equations 11a and 11b, and:

$$\varphi_\theta = \rho_\theta \rightarrow \varphi \tag{15a}$$
$$\psi_\theta = \psi \wedge \rho_\theta \tag{15b}$$

where we can cast the set of all the traces in which all the pairs in θ are equivalent, introduced in Equation 12, as:

$$\rho_\theta = \bigwedge_{x,y \in I_\theta \cup O_\theta, (x,y) \in \theta} \square(x = y) \tag{16}$$

3.4.1. Compatibility and Consistency for LTL A/G Contracts

In Section 3.1.4, we described how it is possible to compute satisfiability for an LTL formula using off-the-shelf tools such as a model checker. To compute consistency and compatibility of an LTL A/G contract, however, satisfiability

is not sufficient. In fact, we need to guarantee that the contract is $I-$ and $O-$receptive, respectively.

We can verify receptiveness of an LTL A/G contract by checking whether its formulas are realizable or not, as described in Section 3.1.4. For instance, to check consistency of a contract $C = (I, O, \varphi, \psi)$, we need to verify that the guarantee ψ is realizable when I is controlled by the environment and O is controlled by the system. Indeed, to be $I-$receptive, the contract needs to accept any sequence generated over variables in I. This corresponds to the realizability game, where the environment is free to choose any assignment to variables in I, to which the system needs to respond accordingly assigning values to variables in O. If ψ is realizable, it means that there exists a model which can properly react to any input from the environment, i.e., there exists a correct implementation of the contract. To check compatibility, conversely, we need to verify that φ is $O-$receptive. This implies that we need to reverse our perspective on who is controlling the variables in I and O. In this case, the realizability game needs to be set such that the set I is controlled by the player, while O is controlled by the adversary. If the formula is realizable, it means that there exists a contract environment that can handle all the outputs of the system implementing the contract. In the experiments performed in the context of this paper, all the contracts in the libraries have been verified for consistency and compatibility using Ratsy [56].

4. Synthesis from Libraries of LTL A/G Contracts

In this section, we describe the synthesis of a composition of elements from a library that satisfies a certain specification. We previously addressed this problem in [21]. In that paper, we focused on the problem of synthesis from libraries of components when minimal assumptions were made on both the formalism used to describe components and the verifier, i.e., an oracle able to determine the correctness of a certain candidate solution. The only requirements imposed on the chosen formalism were being able to compose components and, given two different components, being able to determine if one is the refinement of the other. Furthermore, we only required the verifier to provide a yes/no answer. That synthesis approach relies on generating many candidate solutions, each of which can be independently verified. Specifically, the size of each verification problem depends only on the specification itself and the candidate solution, not on the number of tested candidates.

The method presented in this paper, instead, focuses solely on LTL A/G contracts and uses a model checker as a verifier, leveraging its ability to return counterexample traces. This choice allows us to harness properties unique to LTL and A/G contracts, enabling us to go beyond just observing the Boolean outcome of the refinement checking process. Indeed, we can access more detailed information provided by the underlying model checker, i.e., counterexamples representing complete system runs, and use them to improve the efficiency of the synthesis process. We propose an approach based on CEGIS that utilizes the infinite-length counterexamples generated by the model checker, which we compactly represent as state machines, to guide the synthesis process. This synthesis strategy, in comparison to the one used in [21], is more efficient as it requires fewer calls to the verifier, although generating candidate solutions is more work-intensive. Compared to traditional CEGIS approaches [22, 24, 35],

we had to overcome new technical challenges. For instance, using LTL, inputs and outputs are infinite temporal sequences. What is the best way to represent them? How do we ensure the termination of the algorithms, given that the input space is infinite? We address these questions in this section.

The remainder of this section is organized as follows. Section 4.1 presents the synthesis problem that we address. We propose a solution for a simplified version of the problem in Section 4.2. In Section 4.3 we focus on performance issues, describing, then, our approach to the full problem in Section 4.4. We discuss the detail of our CEGIS algorithm in Section 4.5. In Section 6 we apply the synthesis technique to several case studies.

4.1. Constrained Synthesis from LTL A/G Contracts Libraries (LTL-CSCL)

We first state the problem we aim to solve:

Definition 4.1 (LTL-CSCL problem)**.** *Let $S = (I_S, O_S, \varphi_S, \psi_S)$ be a well-defined LTL A/G contract expressing a system specification, and $L = (\mathcal{Z}, \mathcal{R})$ a library of well-defined LTL A/G contracts, where we assume that $\mathcal{R}$ also imposes constraints over variables in S. The LTL-CSCL problem, then, consists of finding a finite set of contracts $H = \{C_1, \ldots, C_N \mid C_i = (I_i, O_i, \varphi_i, \psi_i) \in \mathcal{Z}\}$, and an interconnect $\theta \subseteq \mathcal{R}$ such that*

$$(C_1 \parallel \cdots \parallel C_N)\theta \preceq S.$$

The notion of well-defined in this definition corresponds to Definition 3.2. Here we leverage definitions and notations that are proper for our contract framework, but the ultimate goal is substantially the same as in [21], i.e., synthesis of a composition of elements from a library such that the specification is satisfied. Here, also, we assume that $\mathcal{R}$ implies all the library-specific constraints, representing the set of all valid connections for L.

4.2. Solving a Simplified Version of the LTL-CSCL Problem

We will develop the solution to the problem in two phases. Here, in the first phase, we build a strategy based on the assumption that $H = \mathcal{Z}$. In other words, for a given synthesis problem, we assume that all the contracts in the library set $\mathcal{Z}$ will be used in the resulting solution set H, if a solution exists. Our focus, initially, is only on how to build θ. Later, in the second phase, we will relax the assumption and show how to solve the problem with $H \subseteq \mathcal{Z}$.

Our solution follows the CEGIS paradigm, under the assumption that we will be able to obtain counterexamples from the verifier over the system variables. We will indicate all the ports in the library as

$$\mathcal{P}^I_{lib} = \bigcup_{i=1}^{|Z|} I_{C1} \tag{17a}$$

$$\mathcal{P}^O_{lib} = \bigcup_{i=1}^{|Z|} O_{C1} \tag{17b}$$

$$\mathcal{P}_{lib} = \mathcal{P}^I_{lib} \cup \mathcal{P}^O_{lib} \tag{17c}$$

and all the ports in the library and the specification as $\mathcal{P}_{lib \cup S}$, defined as

$$\mathcal{P}_{lib \cup S} = \mathcal{P}_{lib} \cup I_S \cup O_S. \tag{18}$$

Example 4.1 (Running Example: A Delay Component)**.** *To help the reader follow the description of our CEGIS formulation, we introduce a running example that we will characterize as we go. The goal of this example is to synthesize a composition to satisfy the following specification*

$$I_{eg} = \{i\} \tag{19a}$$
$$O_{eg} = \{o\} \tag{19b}$$
$$\varphi_{eg} = True \tag{19c}$$
$$\psi_{eg} = \neg o \wedge \bigcirc\neg o \wedge \square(\bigcirc \bigcirc o \leftrightarrow i) \tag{19d}$$

That is, the specification needs the implementation of a double delay on its input. The library, $L_{eg} = (\mathcal{Z}_{eg}, \mathcal{R}_{eg})$, consists of two elements, shown in Table 1, and does not impose any additional constraint through $\mathcal{R}_{eg}$.

Contract	Input Ports	Output Ports	Assumptions	Guarantees
A	x	y	$\varphi_A = True$	$\psi_A = \neg y \wedge \square(\bigcirc o \leftrightarrow i)$
B	v	w, z	$\varphi_B = True$	$\psi_B = \neg w \wedge \square(\bigcirc w \leftrightarrow v) \wedge \square(w \leftrightarrow \neg z)$

Table 1: Library contracts for the running example. Component A implements a single-step delay. Component B is the same as A with the addition of another output, z, that always flips the value of the first output w. Note that these contracts are trivially saturated, as their assumption is always true.

Trivially, a serial composition represented by $\theta_{eg} = \{(x, i), (v, y), (o, w)\}$ satisfies the specification.

4.2.1. LTL-CSCL as a CEGIS instance

To efficiently synthesize an interconnect θ, we use an approach similar to the one introduced in [21], although here all the definitions are applied to contracts rather than generic components. Each port in $\mathcal{P}_{lib \cup S}$ has an associated index, defined according to a function $\mathcal{I}$, and each input port of library contracts, and output port of the specification, has a matching connection variable

$$\mathcal{M} = \left\{ m_p \mid p \in \mathcal{P}^I_{lib} \cup O_S \right\} \tag{20}$$

That is, for a certain θ, if $(x, y) \in \theta$ and x is an input port of a library contract, or output of the specification, then $m_x = \mathcal{I}(y)$. That is, the connection variable of x is equal to the index of y. If $\forall y\colon (x, y) \notin \theta$, then $m_x = -1$. From the assignments defined through $\mathcal{M}$, one can immediately derive the equivalent interconnect $\theta_{\mathcal{M}}$. Note, however, that the opposite does not hold: given a θ, it is not always possible to represent an equivalent set of connections through $\mathcal{M}$ (for instance, with $\mathcal{M}$ we cannot represent the connection of two outputs of library contracts). Given a library $L = (\mathcal{Z}, \mathcal{R})$, going forward, we assume that $\mathcal{R}$ only allows connections that are representable through the variables in $\mathcal{M}$. Then, we can express all the valid connections among contracts, according to $\mathcal{R}$, using the following LTL formulas.

$$\phi_{\mathcal{M}L} = \bigwedge_{C=(I,O,\varphi,\psi)\in Z} \bigwedge_{p\in I} \left(\bigvee_{(p,q)\in\mathcal{R}} [m_p = \mathcal{I}(q)] \wedge \square(p = q) \right) \tag{21}$$

$$\phi_{\mathcal{M}S} = \bigwedge_{s\in O_S} \bigvee_{(s,q)\in\mathcal{R}} ([m_s = \mathcal{I}(q)] \wedge \square(s = q)) \tag{22}$$

Equation 21 encodes the fact that an input variable can be connected to any other port as allowed by $\mathcal{R}$, meaning that the evaluation of the two variables will always match. For each connection variable assignment, we explicitly indicate the behavior we want to observe over the library ports. For instance, if we map the connection variable of a port x to the index of y, that is, $m_x = \mathcal{I}(y)$, then we also expect the formula $\square(x = y)$ to hold, meaning that x and y are connected and have to express the same behavior. Forcing all the input ports to be connected is not, in general, a restrictive assumption. In fact, if a candidate solution works when one of its ports is unconnected, it means that that solution refines the specification no matter the value assigned to that particular port. Fixing the input to the port to be the same as some other port in the design will consist of having a smaller set of possible inputs to that port, thus it will not compromise the outcome in evaluating the solution.

Example 4.2 (Running Example)**.** *Referring to the library from Example 4.1, we encode the valid connections as*

$$\phi_{\mathcal{M}Leg} = (([m_v = \mathcal{I}(i)] \wedge \square(v = i)) \vee ([m_v = \mathcal{I}(y)] \wedge \square(v = y))) \wedge \begin{pmatrix} ([m_x = \mathcal{I}(i)] \wedge \square(x = i)) \vee ([m_x = \mathcal{I}(z)] \wedge \square(x = z)) \vee \\ ([m_x = \mathcal{I}(w)] \wedge \square(x = w)) \end{pmatrix}$$

and

$$\phi_{\mathcal{M}Seg} = \begin{pmatrix} [m_o = \mathcal{I}(z)] \wedge \square(o = z)) \vee ([m_o = \mathcal{I}(w)] \wedge \square(o = w)) \vee \\ ([m_o = \mathcal{I}(y)] \wedge \square(o = y) \end{pmatrix}.$$

Equation 22 encodes the connections of the output ports of the specification, and it is very similar to Equation 21. We refer to the conjunction of the two formulas simply as

$$\phi_{\mathcal{M}} = \phi_{\mathcal{M}L} \wedge \phi_{\mathcal{M}S} \tag{23}$$

Equation 23 requires that each input variable in the library, and output of the specification, has a valid connection, i.e., its connection variable has an index assigned through $\mathcal{M}$.

As, in this phase, a solution includes all the contracts in the library, we also find it useful to define a contract representing the composition of all contracts in the library:

$$C_L = (I_L, O_L, \varphi_L, \psi_L) = \Big|\Big| \{C_i \mid C_i \in \mathcal{Z}\} \tag{24}$$

The contract C_L, while convenient to represent the whole library as a single composition, does not tell us anything about the connections among its constituent contracts. Our goal is to find a definition for $\mathcal{M}$ (hence, a $\theta_{\mathcal{M}}$) such that the following holds

$$\exists \theta_{\mathcal{M}} \colon \phi_{\mathcal{M}} \wedge (C_L \preceq S) \tag{25}$$

meaning that the connections are defined according to $\mathcal{R}$ and C_L refines S.

Considering the constraint implied by Equation 21—no input port of contracts in the library can be unconnected—together with Equation 22, which requires all the specification outputs to be mapped to contracts in the library, we have

that Equations 14a and 14b (part of the refinement conditions) will always be satisfied. Equation 25, then, can be written as $\mathcal{M}$:

$$\exists \theta_{\mathcal{M}} \colon \phi_{\mathcal{M}} \wedge (\varphi_S \rightarrow \varphi_{C_L}) \wedge (\psi_{C_L} \rightarrow \psi_S) \text{ is valid} \tag{26}$$

To be valid, the formula must hold for all the possible executions; we can indicate this condition explicitly as

$$\exists \theta_{\mathcal{M}} \colon \forall \sigma \colon \sigma \models [\phi_{\mathcal{M}} \wedge (\varphi_S \rightarrow \varphi_{C_L}) \wedge (\psi_{C_L} \rightarrow \psi_S)] \tag{27}$$

where σ indicates a behavior, i.e., a trace, over variables in $\mathcal{P}_{lib \cup S}$.

Equation 27, which represents in a concise form the problem described in Definition 4.1, is typical of CEGIS problems, i.e. in the $\exists\forall$ form. As in [23], a solution is articulated in two steps. The first one tries to find a solution that works for all the known counterexamples (solving the $\exists$ part). The second step verifies whether the candidate solution works for all the possible inputs, generating a new counterexample in case it doesn't (solving the $\forall$ part).

4.2.2. Implementation of the CEGIS paradigm for the LTL-CSCL problem

```
1  function General LTL-CSCL:
       Input: library contract C_L = (I_L, O_L, φ_L, ψ_L), constraint formula φ_M,
              specification contract S = (I_S, O_S, φ_S, ψ_S)
       Output: set of connections representing θ, or False
2      E ← {True};                                   // Counterexample set
3      W ← {};                                       // Old candidates
4      while True do
5          model ← checkSAT(φ_syn(C_L, S, φ_M, E, W));
6          if model is unsat then
7              return False;
8          end
9          θ_M ← extractCandidate(model);
10         model ← checkValid(φ_ver(C_L, S, θ_M));
11         if model is valid then
12             return θ_M;
13         end
14         add counterexample from model to E;
15         W ← W ∪ {θ_M};
16     end
17 end
```

Algorithm 1: General description of the implementation of the CEGIS paradigm for the LTL-CSCL problem. Although it captures the essence of our solution, this algorithm will be revisited in the next sections.

Algorithm 1 illustrates the high-level implementation of the CEGIS paradigm for the LTL-CSCL problem. The approach we follow is composed of two main steps, i.e., synthesis and verification. First, in line 5, we check for the satisfiability of a synthesis constraint, i.e., ϕ_{syn}, which we will derive from Equation 27 and the list of counterexamples seen until that point. This operation returns a

model, from which we can extract a candidate set of connections $\theta_{\mathcal{M}}$ (line 9), or `unsat` if no candidate is found. Then, in line 10, the candidate set $\theta_{\mathcal{M}}$ is used to derive a different constraint, $\phi_{\texttt{ver}}$. If $\phi_{\texttt{ver}}$ is valid, then $\theta_{\mathcal{M}}$ represents the correct interconnect. Otherwise, the `checkValid` routine returns a trace that describes a counterexample for the validity check, which is added to the list of counterexamples $\mathcal{E}$. Note that, in line 2, we initialize $\mathcal{E}$ to $\{\mathit{True}\}$, meaning a sequence that is satisfied by any variable assignment. The process terminates when a good candidate is found, or if there is no candidate solution. `checkSAT` and `checkValid` are implemented according to the procedure described in Appendix A.

Although describing the essence of our approach to solve the LTL-CSCL problem, Algorithm 1 is still not complete. For instance, we have no guarantee that the synthesis loop will ever terminate, and the representation of the sequences σ of Equation 27 derived from the counterexamples is still vague. Additionally, we still have to fully describe the constraints $\phi_{\texttt{syn}}$ and $\phi_{\texttt{ver}}$. In the next paragraphs, we will discuss these issues and provide the missing details of our solution.

4.2.3. Handling Infinite Input Sequences

The first problem we need to address is to guarantee that Algorithm 1 terminates. In [24], when the CEGIS approach was first described, the proposed algorithm was guaranteed to terminate because the input space was finite. Thus, in the worst case, for m input variables, eventually the list of counterexamples would contain all the 2^m possible combinations, rendering the synthesis process finite. In our case, however, inputs are *infinite* sequences over a certain set of input variables, meaning that we need a different way to ensure termination.

We solve the problem by keeping track of all the candidate solutions which did not work, together with the counterexamples generated in the verification step. Let $\mathcal{W} = \{\theta_0, \cdots, \theta_k\}$, where θ_i represents an interconnect that does not implement a correct solution, i.e., $C_L\theta_i \not\preceq S$. We can express a constraint that enumerates all the wrong candidates and prevents them from appearing again as

$$\phi_{\mathcal{W}} = \neg \bigvee_{\theta_i \in \mathcal{W}} \bigwedge_{(x,y)\in\theta_i} (\square(x = y)) \tag{28}$$

To prevent the generation of old candidates, then, we will need to consider Equation 28 when defining the synthesis constraint $\phi_{\texttt{syn}}$. The addition of $\phi_{\mathcal{W}}$ to $\phi_{\texttt{syn}}$ ensures the termination of the algorithm as the number of possible candidates is finite, albeit potentially very large.

The other problem that arises from having infinite sequences as counterexamples is finding a suitable way to represent them. In Equation 27, in fact, σ refers to sequences, but there is no additional information on how the sequences are represented.

We decided to represent each behavior σ as the output of an *ad-hoc* state machine. This state machine, then, can be added to the model-checking problem that verifies the validity of $\phi_{\texttt{syn}}$. In general, we can represent the validity check for an LTL formula as a language containment problem. Thus, on the one hand, we have an unconstrained state machine, F, that is able to generate any trace $\mathcal{L}(F) = \mathcal{T}$; on the other hand, we have the LTL formula we want to check. For instance, to check the validity or satisfiability of a formula $\forall\sigma\colon \sigma \models \phi$, or simply

ϕ, we need to model-check:

$$\mathcal{T} \subseteq \mathcal{L}(\phi) \text{ for validity} \tag{29a}$$
$$\mathcal{T} \subseteq \mathcal{L}(\neg\phi) \text{ for satisfiability} \tag{29b}$$

Equations 29 helps us understand how to relate traces and LTL formulas, and can be checked by a model checker. In Section Appendix A.1.2, we discussed how all counterexamples generated by a model checker (NuXmv in our case) are either finite or lasso-shaped. For such counterexamples, we can always find a corresponding finite-state machine that can produce the specific sequence they describe.

```
1  function TraceGenerator:
       Input: counterexample trace c, set of variables 𝒱
       Output: state machine M generating trace c
2      S : list;                                   // list of states
3      T : list;                                   // list of transitions
4      D : hashtable;      // map of variable evaluations for each
         state
5      n ← numberOfStates(c);
6      l ← loopIndex(c);      // l ∈ {−1} ∪ {1, ··· , n}.  −1 means no
         loop
7      i ← 1;
8      S ← add initial state i;
9      D[i] ← evaluateTraceAt(𝒱,c,i);
10     for i ∈ {1, ··· , n} do
11         S ← add state i;
12         D[i] ← evaluateTraceAt(𝒱,c,i);
13         T ← add transition from state i −1 to i;
14         if i = n then
15             if l = −1 then // we add a generic state with no
                 evaluation
16                 S ← add state i + 1;
17                 T ← add transition from state i to i + 1;
18                 T ← add transition from state i + 1 to i + 1;
19             else // there is a loop, we go back to that state
20                 T ← add transition from state i to i + 1;
21             end
22         end
23     end
24     return M = (S, D, T)
25 end
```

Algorithm 2: Constructs a finite-state machine able to generate the same sequence of symbols described by a counterexample trace, as described in Section Appendix A.1.2. The resulting state machine can then immediately be encoded as an SMV module.

Algorithm 2 illustrates how to build a state machine generating a certain counterexample σ. We assume we have available some basic primitives, i.e.,

(i) `numberOfStates`, which returns the total number of states of a certain counterexample, (ii)`loopIndex`, which indicates the location of the loop in the sequence, or returns -1 if no loop is present, and (iii) `evaluateTraceAt`, which returns the evaluation of a certain set of variables at a given step of the counterexample. Once a state machine is obtained, its description as a module for the model checker is straightforward. In the case of NuXmv, models are expressed in the SMV language [69].

At this point, we have all the elements to derive an initial version of the two formulas $\phi_{\texttt{syn}}$ and $\phi_{\texttt{ver}}$ introduced in Algorithm 1. As we will revisit these formulas later, we refer to them here as $\phi'_{\texttt{syn}}$ and $\phi'_{\texttt{ver}}$. In $\phi'_{\texttt{syn}}$, we want to find one assignment for $\mathcal{M}$ which satisfies all the counterexamples seen until a certain iteration. While in Equation 27 we are interested in the validity of the formula, here we are only looking for one satisfiable assignment. We expect that the pre-conditions are met, i.e., $\phi_{\mathcal{M}}$, that we do not include discarded candidates ($\phi_{\mathcal{W}}$), and that the contract obligations are satisfied, meaning, as explained in Section 3.2.1, that we expect the correct behavior of the contract under correct assumptions. Thus, $\phi'_{\texttt{syn}}$ becomes:

$$\begin{aligned}\phi'_{\texttt{syn}} = \exists\theta_{\mathcal{M}} \colon \forall\sigma_i \in \mathcal{E} \colon \{\sigma_i \models [\phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge (\varphi_S \wedge \psi_S) \\ \wedge (\varphi_{C_L} \wedge \psi_{C_L})]\}[x/x_i]_{x\in\mathcal{P}_{lib\cup S}}\end{aligned} \tag{30}$$

where $\mathcal{E}$ is the set containing the observed counterexamples, over the specification inputs, introduced in Algorithm 1. Here, we only care about specification inputs because that is how an adversarial environment interacts with candidate solutions. Finally, $[x/x_i]_{x\in\mathcal{P}_{lib\cup S}}$ indicates the syntactical renaming of the variables in $\mathcal{P}_{lib\cup S}$ with fresh variables in the enclosed formulas and trace. The equation above can be solved for satisfiability by model checking that:

$$\bigwedge_{\sigma_i\in\mathcal{E}} \{\mathcal{L}(F_{\sigma i}) \subseteq \mathcal{L}(\neg[\phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge (\varphi_S \wedge \psi_S) \wedge (\varphi_{C_L} \wedge \psi_{C_L})])\}[x/x_i]_{x\in\mathcal{P}_{lib\cup S}} \tag{31}$$

where $F_{\sigma i}$ is a state machine generated from σ_i according to Algorithm 2. Note that, in this case, only the variables in $\mathcal{P}_{lib\cup S}$ are renamed. The connection variables and indices represented by $\mathcal{M}$ and $\mathcal{I}$ within $\phi_{\mathcal{M}}$ (cf. Equation 21) are not renamed across the different terms.

It is critical to understand the importance of using contracts obligations here instead of the usual refinement implications. If we had used those, i.e., $(\varphi_S \rightarrow \varphi_{C_L}) \wedge (\psi_{C_L} \rightarrow \psi_S)$, then the formula could be satisfied, for instance, simply by falsifying ψ_{C_L}, meaning that the library contracts did not behave according to their guarantees, or by falsifying φ_S, independently of the actual choice of connections $\mathcal{M}$.

For $\phi'_{\texttt{ver}}$, instead, we are interested in proving that the candidate $\theta_{\mathcal{M}}$, derived by the assignments of $\mathcal{M}$ in $\phi'_{\texttt{syn}}$, is the correct one in all the scenarios. Hence, now we are interested in proving the formula valid. In this case, the formula only needs to verify that refinement always holds:

$$\phi'_{\texttt{ver}} = C_L\theta_{\mathcal{M}} \preceq S = (\varphi_S \rightarrow \varphi_{C_L\theta_{\mathcal{M}}}) \wedge (\psi_{C_L\theta_{\mathcal{M}}} \rightarrow \psi_S) \tag{32}$$

Although, at this point, we can see that using $\phi'_{\texttt{syn}}$ and $\phi'_{\texttt{ver}}$ as synthesis and verification constraints in Algorithm 1 will make it terminate and yield a correct refinement, if one is found, there are still some issues that can affect the overall

performance of the synthesis process. In the next sections, we will discuss them and show how to improve the algorithm.

Example 4.3 (Running Example)**.** *Consider the specification and library introduced in Example 4.1. During the first iteration of Algorithm 1, we can verify* $\phi'_{syn_{eg1}}$ *(the numeric subscript indicates that we are in the first iteration of the algorithm) by model checking that the following formula, derived from Equation 31, is not empty*

$$\mathcal{T} \subseteq \mathcal{L}(\neg\phi_{eg}) = \mathcal{L}(\neg[\phi_{\mathcal{M}eg} \wedge \phi_{\mathcal{W}eg} \wedge (\varphi_{Seg} \wedge \psi_{Seg}) \wedge (\varphi_{C_{Leg}} \wedge \psi_{C_{Leg}})])$$

where $\phi_{\mathcal{M}eg} = \phi_{\mathcal{M}Leg} \wedge \phi_{\mathcal{M}Seg}$ *as defined in Example 4.2,* $\phi_{\mathcal{W}eg} = True$ *because we don't have any incorrect candidates yet,* $\psi_{C_{Leg}} = \psi A \wedge \psi B$ *and* $\varphi_{C_{Leg}} = \psi_{C_{Leg}} \rightarrow (\varphi_A \wedge \varphi_B)$*, representing guarantees and assumptions of the library contract. As this is the first iteration of the synthesis process, we use* $\mathcal{T}$ *in the equation above to represent a state machine that can generate any trace. For the same reason, there is no need to explicitly rename variables as indicated by* $[x/x_i]_{x\in\mathcal{P}_{lib\cup S}}$ *in Equation 31.*

The model checker returns a counterexample where ϕ_{eg} *holds. From there, we can infer the evaluation of* $\mathcal{M}_{eg}$ *and, therefore, the connections that make the contracts satisfy* $\phi'_{syn_{eg}}$*. Let us assume that we inferred the following connections* $\mathcal{M}_1$

$$m_{v1} = \mathcal{I}(i), m_{x1} = \mathcal{I}(i), m_{o1} = \mathcal{I}(w)$$

That is, contract $\boldsymbol{B}$ *controls the specification output* o *and takes its input from the specification input* i*. Contract* $\boldsymbol{A}$*, instead, takes its input from contract* $\boldsymbol{B}$ *but its output is left floating.*

Solving $\phi'_{syn_{eg1}}$ *tells us that there is at least one case where the solution works—looking at the contract library, indeed, the reader can easily verify that this candidate solution only works in a specific case, when* i *is always false. The algorithm then proceeds to verify whether the candidate solution always holds. It does so by checking the validity of* $\phi'_{ver_{eg1}}$*, derived from Equation 32:*

$$\phi'_{ver_{eg1}} = (\varphi_{eg} \rightarrow \varphi_{C_{Leg}\theta_{\mathcal{M}1}}) \wedge (\psi_{C_{Leg}\theta_{\mathcal{M}1}} \rightarrow \psi_{eg})$$

In the formula above, we refer to the composition of contracts $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *with the application of the interconnect* $\theta_{\mathcal{M}1}$*,* $C_{Leg}\theta_{\mathcal{M}1}$*, as described in Section 3.4*

$$\begin{aligned}
&I\theta_{\mathcal{M}1} = \{i\}\\
&O\theta_{\mathcal{M}1} = \{y, w, z, v, x, o\}\\
&\varphi_{C_{Leg}\theta_{\mathcal{M}1}} = \rho_{\theta\mathcal{M}1} \rightarrow \varphi_{C_{Leg}}\\
&\psi_{C_{Leg}\theta_{\mathcal{M}1}} = \psi_{C_{Leg}} \wedge \rho_{\theta\mathcal{M}1}
\end{aligned}$$

where $\rho_{\theta\mathcal{M}1} = \square(v = i) \wedge \square(x = w) \wedge \square(o = w)$*, according to Equation 16.*

Model-checking $\phi'_{ver_{eg1}}$ *shows that we did not find a general solution: the candidate fails if, for instance,* i *is always true. We call this first counterexample* σ_1*. Before proceeding with a new iteration, as Algorithm 1 shows, we keep track of* σ_1 *and* $\theta\mathcal{M}1$*.* σ_1 *is used to create a state machine able to generate that sequence, according to Algorithm 2.* $\theta\mathcal{M}1$*, instead, is used to update* $\phi_{\mathcal{W}eg}$ *by adding the connections of the incorrect candidate, according to Equation 28*

$$\phi_{\mathcal{W}eg} = \neg\left(\square(v = i) \wedge \square(x = w) \wedge \square(o = w)\right)$$

We are now ready to perform a new iteration of Algorithm 1, by verifying $\phi'_{syn_{eg2}}$ *as*

$$\begin{aligned}&\Big(\mathcal{T} \subseteq \mathcal{L}(\neg[\phi_{\mathcal{M}eg} \wedge \phi_{\mathcal{W}eg} \wedge (\varphi_{Seg} \wedge \psi_{Seg}) \wedge (\varphi_{C_{Leg}} \wedge \psi_{C_{Leg}})])[x/x_0]_{x\in\mathcal{P}_{lib\cup S}}\Big) \wedge \\ &\Big(\mathcal{L}(F_{\sigma 1}) \subseteq \mathcal{L}(\neg[\phi_{\mathcal{M}eg} \wedge \phi_{\mathcal{W}eg} \wedge (\varphi_{Seg} \wedge \psi_{Seg}) \wedge (\varphi_{C_{Leg}} \wedge \psi_{C_{Leg}})])[x/x_1]_{x\in\mathcal{P}_{lib\cup S}}\Big)\end{aligned} \tag{33}$$

In practice, once we have at least one concrete counterexample, it is not necessary to also verify the containment of the language in $\mathcal{T}$*, but here it is useful to have it to describe how the formulas look after variable renaming. For simplicity, let us only look at a subset of* $\phi_{\mathcal{M}eg}$ *from the formula above,* $([m_v = \mathcal{I}(i)] \wedge \Box(v = i))$*, that includes variables from* $\mathcal{P}_{lib\cup S}$ *but also* $\mathcal{M}$*. We have that*

$$\begin{aligned}&(([m_v = \mathcal{I}(i)] \wedge \Box(v = i))[x/x_0]_{x\in\mathcal{P}_{lib\cup S}}) \wedge \\ &(([m_v = \mathcal{I}(i)] \wedge \Box(v = i))[x/x_1]_{x\in\mathcal{P}_{lib\cup S}})\end{aligned}$$

is equivalent to

$$([m_v = \mathcal{I}(i)] \wedge \Box(v_0 = i_0)) \wedge ([m_v = \mathcal{I}(i)] \wedge \Box(v_1 = i_1)).$$

Note that while the variables v *and* i *are renamed, variable* m_v *and index* $\mathcal{I}(i)$ *stay the same across the formulas. In this way, we can effectively have several instances of the same model at once. While the connection variables do not change between instances, the model checker can evaluate different candidate behaviors, one per counterexample.*

When verifying $\phi'_{syn_{eg2}}$ *through* (33) *above, the model checker proposes the following connections* $\mathcal{M}_2$*:*

$$m_{v2} = \mathcal{I}(i), m_{x2} = \mathcal{I}(w), m_{o2} = \mathcal{I}(y).$$

That is, now contract $\boldsymbol{B}$ *gets the specification input and feeds contract* $\boldsymbol{A}$ *through* w*. contract* $\boldsymbol{A}$*, in turn, controls the specification output. We can observe that now* σ_1 *will not be able to break this candidate. Indeed, not only does this candidate work with the previous counterexample, but it is also a general solution. We can verify it by checking the validity of*

$$\phi'_{ver_{eg2}} = (\varphi_{eg} \rightarrow \varphi_{C_{Leg}\theta_{\mathcal{M}2}}) \wedge (\psi_{C_{Leg}\theta_{\mathcal{M}2}} \rightarrow \psi_{eg})$$

where $C_{Leg}\theta_{\mathcal{M}2}$ *is built same as* $C_{Leg}\theta_{\mathcal{M}1}$ *above but considering* $\mathcal{M}_2$ *instead.*

4.3. Performance Considerations

Consider a library with n contracts with p input and output ports, which can accept and generate any behavior over those ports, i.e., their assumption and guarantee are always true: $\varphi_i = \psi_i = \mathit{True}$.

Let S be a generic specification such that $\varphi_S, \psi_S \neq \mathit{True}$, also with p input and output ports, and try to observe how the CEGIS Algorithm 1 would work in such case, with $\phi'_{\mathtt{syn}}$ and $\phi'_{\mathtt{ver}}$ as defined in the previous section. First, in line 5, the algorithm tries to satisfy Equation 30, finding a candidate connection $\theta_{\mathcal{M}}$. Any of the n contracts is able to generate a sequence satisfying $\phi'_{\mathtt{syn}}$, as they can generate any trace. When verifying Equation 32, in line 10, the model checker shows that indeed that candidate is not correct, as the candidate solution can also generate traces such that $\psi_{C_L\theta_c} \not\rightarrow \psi_S$. `checkValid` returns a counterexample

on the inputs of S showing that the refinement does not hold. This could be any trace. Then, the cycle repeats, keeping track of the bad candidate $\theta_{\mathcal{M}}$ through $\varphi_{\mathcal{W}}$. Once again, any connection $\theta'_{\mathcal{M}} \neq \theta_{\mathcal{M}}$ could satisfy S for that specific counterexample, and so on.

It is easy to see that, for this library, counterexamples are not really helping in solving the problem. The algorithm will eventually terminate, after having tried all the possible $2^{\frac{np(np-1)}{2}}$ candidates. It will do so slowly, as for each counterexample the formula in Equation 31 grows, making the satisfiability problem more and more complex.

This is, of course, an extreme scenario, but it highlights why having nondeterministic contracts in the library is, in general, not desirable.

```
1  function IsDeterministic:
       Input: contract C = (I, O, φ, ψ)
       Output: True if C is deterministic, False otherwise
2      C' → copy of C;                    // as defined in Section 3.2.1
3      ρ_I → ⋀_{i∈I} □(i = C'.i);
4      ρ_O → ⋀_{o∈O} □(o = C'.o);
5      m = checkValid((φ ∧ ψ) ∧ (φ' ∧ ψ') ∧ ρ_I → ρ_O);
6      if m is valid then
7          return True
8      else
9          return False
10     end
11 end
```

Algorithm 3: Algorithm to check if a contract C is deterministic. It does so by verifying that, when given the same input sequence, under the expected assumption and guarantee, C and its copy C' will always return the same output.

Algorithm 3 shows a procedure that can detect whether a contract C is nondeterministic, meaning that, for a certain sequence over its input ports that satisfies its assumption, at least two distinct sequences over its output ports satisfy its guarantee. It does so by creating a copy C' of C, and verifying whether they can generate different output sequences given the same inputs. The algorithm is sound and complete. Indeed, if model-checking the formula in Line 5 of the algorithm shows that it is not valid, then a counterexample will be generated, too. In the resulting trace, then, the left-hand side of the formula will be true (hence identical inputs), but the right-hand side won't hold, showing two different output sequences. If no counterexample can be found, then C is deterministic by definition. Reversing our reasoning, if C is deterministic, then model checking the formula in Line 5 of the algorithm will necessarily show that the formula is valid (no counterexample can be found). If C is non-deterministic, then there exists at least a trace in which the inputs are the same, but the outputs are not. This trace is, by definition, a counterexample, hence it will also be returned by the model checker. This algorithm does not affect the formulation of our synthesis methodology but can be used while defining the contract library to make sure all the contracts have deterministic behaviors.

4.3.1. Nondeterminism, Cycles, and Depth

Avoiding nondeterministic contracts, however, it's not enough to guarantee the absence of nondeterministic behaviors when connecting contracts. Consider, for instance, the following example.

Example 4.4 (Simple Cycle)**.** *Let $C = (\{a\}, \{b\}, True, \square(b = a))$ be a contract that simply relays whatever input is given to it, and $\theta = \{(a, b)\}$ be an interconnect which implies a connection between input and output of C. $C\theta$, then, is a nondeterministic contract as its guarantee will be satisfied no matter the value of the output port b.*

Example 4.4 shows that cycles can be problematic, as they could be a source of nondeterministic behaviors during the synthesis process. This is not going to affect the final result, but it can have a substantial impact on the overall performance.

Additionally, Example 3.1 shows that creating a cycle can yield an inconsistent contract. Formally, an inconsistent composition could be a refinement of a specification because its guarantee is empty. Allowing cycles means that each solution should be checked for consistency, which could be a prohibitive task as it requires verifying the realizability of the contract's formulas (cf. Section 3.1.4).

Therefore, we explicitly avoid cycles in the contract compositions. To enforce this constraint, we consider an additional set of variables, one for each contract in the library called location variables. We indicated them as

$$\mathcal{V}_L = \{l_C \mid C \in \mathcal{Z}\} \tag{34}$$

We do not directly assign any value to variables in $\mathcal{V}_L$, but require the value of a contract's location variable to be greater than the location variables of the contracts that feed it its inputs:

$$\phi_{LL} = \bigwedge_{\substack{C_1=(I_1,O_1,\\ \varphi_1,\psi_1)\in Z}} \bigwedge_{p\in I_1} \bigwedge_{\substack{C_2=(I_2,O_2,\\ \varphi_2,\psi_2)\in Z}} \bigwedge_{q\in O_2} [(m_p = \mathcal{I}(q)) \rightarrow (l_{C1} > l_{C2})] \tag{35}$$

and greater than zero if mapped to the specification inputs:

$$\phi_{LS} = \bigwedge_{C_1=(I_1,O_1,\varphi_1,\psi_1)\in Z} \bigwedge_{p\in I_1} \bigwedge_{q\in O_S} [(m_p = \mathcal{I}(q)) \rightarrow (l_{C1} > 0)] \tag{36}$$

Then, we indicate as ϕ_L the conjunction of Equations 35 and 36:

$$\phi_L = \phi_{LL} \wedge \phi_{LS} \tag{37}$$

The addition of location variables for contracts in the library does not affect too much the overall size of the synthesis formulas, as the number of such variables is much smaller than the variables encoded through $\mathcal{M}$, which represents all the ports in the library. Additionally, we can use the location variables to limit the maximum depth of a solution by imposing an upper bound to the variables in $\mathcal{V}_L$:

$$\phi_D(d) = \bigwedge_{l\in\mathcal{V}_L} l \leq d \tag{38}$$

where d is the desired maximum depth. Moreover, given a depth d, we can explore the solution space in incremental steps from depth 1 to d. This guarantees that, when a solution is found, it will also have minimal depth.

Later, we will discuss how to integrate the additional constraints described in this section with the synthesis technique introduced with Algorithm 1.

4.4. Addressing the Full LTL-CSCL Problem

In this section, we will relax the assumption $H = \mathcal{Z}$ that we made in Section 4.2, thus allowing solutions where only a subset of the library is used, $H \subseteq \mathcal{Z}$. In general, having too many contracts in a candidate solution could be problematic as, even if they are not connected to any other contract, they might introduce additional assumptions which will prevent the refinement between the specification and the composition to hold. Consider, for instance, the following example, where we try to find a composition satisfying a specification under the assumption $H = \mathcal{Z}$.

Example 4.5 (Assumptions Too Restrictive)**.** *Let $S = (\{x\}, \{y\}, True, \Diamond y)$ be a contract representing a specification. Let also $L = (\mathcal{Z}, \mathcal{R})$ be a library where $\mathcal{Z}$ contains only two contracts, $C_1 = (\emptyset, \{b\}, True, \Diamond b)$ and $C_2 = (\{a\}, \emptyset, \Box\neg a, True)$, where $\mathcal{R}$ allows any connections. Note that C_2 does nothing, as it has no output ports.*

One can easily see that, indeed, S can be refined by C_1 by mapping y to b. According to the assumption $H = \mathcal{Z}$, however, we need to include also C_2 in the resulting composition. Thus, if a is mapped to x, then the composition becomes

$$(C_1 \parallel C_2)\theta = (\{x\}, \{a, y, b\}, \\ \Box(a = x \wedge b = y) \rightarrow (\Diamond b \rightarrow \Box\neg a), \Box(a = x \wedge b = y) \wedge \Diamond b)$$

and the refinement with S holds if and only if the following are both valid:

$$True \rightarrow [\Box(a = x \wedge b = y) \rightarrow (\Diamond b \rightarrow \Box\neg a)] \\ [\Box(a = x \wedge b = y) \wedge \Diamond b] \rightarrow \Diamond y$$

The first formula, however, is not valid, thus the refinement does not hold.

If a is connected to b, instead, then the composition (where the ports have been renamed according to the new connection) is

$$(C_1 \parallel C_2)\theta = (\emptyset, \{a, y, b\}, \\ \Box(a = b \wedge b = y) \rightarrow (\Diamond b \rightarrow \Box\neg a), \Box(a = b \wedge b = y) \wedge \Diamond b)$$

and the refinement with S is verified if and only if the following are both valid:

$$True \rightarrow [\Box(a = b \wedge b = y) \rightarrow (\Diamond b \rightarrow \Box\neg a)] \\ [\Box(a = b \wedge b = y) \wedge \Diamond b] \rightarrow \Diamond y$$

The refinement, again, does not hold as the first formula is not valid.

Although unconnected inputs are not allowed in our approach, even leaving the variable a unconnected would result in a non-valid refinement formula, as one can immediately check.

Hence, under the assumption $H = \mathcal{Z}$, no composition refining S can be found.

Example 4.5 shows why, sometimes, requiring a solution to a synthesis problem to include all the contracts in the library is not a good idea. To solve this problem, we give the ability to the model checker to select which contracts

to include in a candidate solution. We do so by using the contract location variables introduced in Equation 34; if any of such variables is set to 0, then it means that its associated contract does not participate in the final composition. Thus, we can selectively ignore its assumption and guarantee by parameterizing it according to its location variable:

$$C^{l_C} = (I, O, (l_C > 0) \rightarrow \varphi, (l_C > 0) \rightarrow \psi) \tag{39}$$

The parameterized contract will then be identical to C if $l_C > 0$, or it will be trivial ($\varphi = \psi = \mathit{True}$). In this way, the solver will not be forced to satisfy assumptions and guarantees of unused contracts. We indicate the composition of all the parameterized library contracts (cf. Equation 24) as

$$C_L^{\mathcal{V}_L} = (I_L, O_L, \varphi_L^{\mathcal{V}_L}, \psi_L^{\mathcal{V}_L}) = \Big|\Big| \{C_i^{l_{Ci}} \mid C_i \in \mathcal{Z}\} \tag{40}$$

Additionally, we require that, if a contract is not used, it cannot feed any other contract:

$$\phi_C = \bigwedge_{[C_1=(I_1,O_1,\varphi_1,\psi_1)\in Z]} \bigwedge_{[p\in I_1]} \bigwedge_{[C_2=(I_2,O_2,\varphi_2,\psi_2)\in Z]} \bigwedge_{[q\in O_2]} [(l_{C2} = 0) \rightarrow (m_p \neq \mathcal{I}(q))] \tag{41}$$

We enforce this additional constraint because otherwise we would allow nondeterministic inputs, which are not desirable, as discussed in Section 4.3.1.

4.5. Putting It All Together: Revisited LTL-CSCL

In this section, we revisit the initial synthesis and verification constraints, $\phi'_{\texttt{syn}}$ and $\phi'_{\texttt{ver}}$, introduced in Equations 30 and 32, to include all the improvements discussed in Sections 4.3 and 4.4. We start from the synthesis constraint, which now becomes:

$$\begin{aligned} \phi_{\texttt{syn}}(C_L, S, \phi_{\mathcal{M}}, \mathcal{E}, \mathcal{W}, d) = \exists \theta_{\mathcal{M}}, \mathcal{V}_L \colon \forall \sigma_i \in \mathcal{E} \colon \\ \{\sigma_i \models \phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge \phi_L \wedge \phi_D(d) \wedge \phi_C \wedge \\ (\varphi_S \wedge \psi_S) \wedge (\varphi_{C_L}(\mathcal{V}_L) \wedge \psi_{C_L}(\mathcal{V}_L))\}[x/x_i]_{x\in\mathcal{P}_{lib}\cup S} \end{aligned} \tag{42}$$

where σ_i represents the sequence generated by the *ad-hoc* state machine built according to Algorithm 2 representing the i-th counterexample. The formula above can be checked for satisfiability by solving a problem similar to the one in Equation 31. It is worth noticing how this time we are also looking for an assignment of the variables in $\mathcal{V}_L$, which parameterizes the library contract C_L. The addition of all the extra terms, here, is not a problem. A candidate $\theta_{\mathcal{M}}$ satisfying Equation 42 will also satisfy Equation 27, which represents our original synthesis goal.

The verification constraint, instead, does not change; it requires us to consider only the contracts used in the candidate solution rather than the whole library, according to the candidate set H inferred by the assignments to the variables $\mathcal{V}_L$ in $\phi_{\texttt{syn}}$. We indicate the candidate composition as $C_H = (I_H, O_H, \varphi_H, \psi_H) = ||\ \{C_i \mid C_i \in H\}$, and the verification constraint includes it as:

$$\begin{aligned} \phi_{\texttt{ver}}(C_H, S, \theta_{\mathcal{M}}) &= \phi'_{\texttt{ver}}(C_H, S, \theta_{\mathcal{M}}) \\ &= (\varphi_S \rightarrow \varphi_{C_H\theta_{\mathcal{M}}}) \wedge (\psi_{C_H\theta_{\mathcal{M}}} \rightarrow \psi_S) \end{aligned} \tag{43}$$

where $\theta_{\mathcal{M}}$ is the interconnect obtained by solving $\phi_{\texttt{syn}}$.

We can now provide a detailed representation of the LTL-CSCL algorithm, described in Algorithm 4, which takes into account the newly added constraints and the incremental depth strategy.

```
function Full_LTL-CSCL:
    Input: library contract C_L = (I_L, O_L, φ_L, ψ_L), constraint formula φ_M,
           specification contract S = (I_S, O_S, φ_S, ψ_S), maximum solution
           depth D
    Output: set of contracts H ⊆ 𝒵 and set of connections representing θ,
            or False
    d = 1;
    ℰ ← {𝒯};                                   // Counterexample set
    𝒲 ← {};                                    // Old candidates
    while d ≤ D do
        while True do
            model ← checkSAT(φ_syn(C_L, S, φ_M, ℰ, 𝒲, d));
            if model is unsat then
                d = d + 1;
            end
            (H, θ_M) ← extractCandidate(model);
            model ← checkValid(φ_ver(C_H, S, θ_M));
            if model is valid then
                return (H, θ_M);
            end
            σ ← extractCounterexample(model);
            F_σ ← TraceGenerator(σ);
            ℰ ← ℰ ∪{F_σ};
            𝒲 ← 𝒲 ∪ {θ_M};
        end
    end
    return False;
end
```

Algorithm 4: Description of the algorithm solving the LTL-CSCL problem including all the performance considerations discussed in Section 4.3.

4.6. Evaluation

We tested the synthesis algorithm on three different scenarios. The first is inspired by the field of embedded design, and it is about the synthesis of the architecture of a Serial Peripheral Interface (SPI) controller for an Analog-to-Digital converter (ADC); the second is the design of a brushless DC electric motor (BLDC); and the third problem is the design of the electrical power distribution system (EPS) of an aircraft. The details of our analysis are reported in Section 6.

5. Specification Decomposition for Synthesis from Libraries of LTL A/G Contracts

In this section, we present a way to increase the scalability of synthesis from libraries of LTL A/G contracts. Given a specification, also described by an LTL A/G contract, we show how to efficiently partition the synthesis problem into several simpler sub-problems, which can be solved independently. We do so by analyzing counterexamples generated by a model checker when asked to verify the validity of a properly crafted formula. Note that, while we use similar techniques, in principle, to the ones used for synthesis, here the objective is different. The result is an algorithm able to generate a set of sub-specifications that are as "small" as possible, only requiring a number of operations quadratic in the number of variables of the original contract. These smaller specifications, then, can be independently synthesized. This section leverages the property of independent implementability of contracts, which derives from the notions of refinement and composition. For instance, let C, C_1, C_2 be contracts such that $C_1 \parallel C_2 \preceq C$. Let also C_1', C_2' be contracts such that $C_1' \preceq C_1$, and $C_2' \preceq C_2$. Then $C_1' \parallel C_2' \preceq C$ holds, too.

We will refer to LTL A/G contracts and libraries as introduced in Section 3 and discussed further in Section 4, e.g., an LTL A/G contract is indicated as $C = (I, O, \varphi, \psi)$, and a library is a pair $L = (\mathcal{Z}, \mathcal{R})$. The system specification $S = (I_S, O_S, \varphi_S, \psi_S)$ that needs to be synthesized is also an LTL A/G contract.

This section builds on top of the concepts, problems, and solutions discussed in Section 4. The algorithm and results presented in this paper differ slightly from those in [26]. Theorem 5.1 is stronger, and Algorithm 5 does not limit the number of variables used in a counterexample. We have also removed the claim of completeness, which was found to be incorrect.

5.1. Decomposing Contracts

Given a system specification expressed as an LTL A/G contract, our objective is to decompose it into several sub-specifications (or projections), to simplify the synthesis problem in Definition 4.1. In this section, we show how to formally describe these projections and how it is possible to treat them independently while guaranteeing the satisfaction of the original specification. Thus, given a contract $C = (I, O, \varphi, \psi)$, we want to find a composition of contracts $C' = C_1' \parallel \cdots \parallel C_n'$ such that

$$C_1' \parallel \cdots \parallel C_n' \preceq C. \tag{44}$$

Note that, compared to the problem addressed in section 4, here each contract C_i' does not need to come from a library, but it will become a specification for a different synthesis task. The independent implementability property of contracts,

introduced in Section 3.2.1, guarantees that refining the single contracts on the left-hand side of Equation 44 will yield a valid refinement for C.

Ideally, C'_i will need to be simpler to synthesize than the original contract C. The complexity of synthesis, as discussed in [21], ultimately depends on the number of connections that the synthesizer needs to establish. Hence, it is desirable that the synthesis problem for each C'_i results in fewer unknown connections, when compared to the original specification C.

Another desirable property of such decomposition is that it will need to maintain intact the solution space. That is, any solution that could be found by synthesizing C should also be found by independently synthesizing each C'_i. Thus, we would like contracts C'_is to be as permissive as possible, meaning that their guarantees should be the weakest, and their assumptions the strongest, while still holding the refinement in Equation 44. Finally, the decomposition process should always be successful, meaning that, in the worst case, we should trivially obtain the same contract C.

In the next sections, we will introduce the notion of projection for LTL A/G contracts and will show how it can be used to define and solve the problem of contract decomposition.

5.1.1. Projection for LTL A/G Contracts

According to the discussion in the previous section, each contract resulting from the decomposition of a specification C should be simpler to synthesize than C. We discussed how this boils down to reducing the number of unknown connections that a synthesizer needs to solve. Here, we introduce the concept of *projection* to characterize each C'_i of Equation 44. Projections will be useful to express our intent to have each C'_i define only some properties of the specification C, which is then realized as the composition of all projections.

Defining a projection for an A/G contract C, intuitively, means describing a new contract over a subset of its variables. The new assumptions and guarantees, then, will need to be defined only over those variables exposing all the behaviors that could be observed in C by looking at the subset of variables. Unfortunately, Wolper [70] proves that LTL formulas are not, in general, closed under projection. Therefore, for LTL A/G contracts, we cannot define the projection operation by simply taking the projection of their assumption and guarantee formulas. Instead, we provide a definition of projection for LTL A/G contracts that works by partitioning the contract output variables, without the need of projecting LTL formulas. The reasons behind the choice of defining the projection only over output variables will be clarified later.

Definition 5.1 (Projection of LTL A/G Contracts)**.** *Given an LTL A/G contract $C = (I, O, \varphi, \psi)$ and a subset of its output variables $V \subseteq O$, its projection with respect to V is a contract*

$$\Pi_V(C) = C^c \theta_V \tag{45}$$

where C^c is a copy of C with fresh variables as in Definition 3.4, and θ_V specifies the following connections:

$$\forall x \in I : (C^c.x, x) \in \theta_V \tag{46}$$

$$\forall x \in O : x \in V \Leftrightarrow (C^c.x, x) \in \theta_V \tag{47}$$

Thus, $\Pi_V(C)$ shares with C all input variables and the variables in V, while all the other output variables are left *unconnected*.

Given a projection $\Pi_V(C)$, we call it valid if and only if

$$\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C \tag{48}$$

where $\bar{V} = \{p \mid p \in O \setminus V\}$ contains all the output variables of C which are not in V. That is, $\Pi_{\bar{V}}(C)$ complements $\Pi_V(C)$ with respect to the output ports of C.

Example 5.1 (Not all projections are valid)**.** *Consider the contract*

$$C = (\emptyset, \{a, b\}, \varphi = \mathit{True}, \psi = \Box(a \vee b))$$

and the set $X = \{a\}$*. The projection associated with* X *is*

$$\Pi_X(C) = (\emptyset, \{a, a_0, b_0\}, \mathit{True}, \Box(a = a_0) \wedge \Box(a_0 \vee b_0))^4$$

while its complement is

$$\Pi_{\bar{X}}(C) = (\emptyset, \{b, a_1, b_1\}, \mathit{True}, \Box(b = b_1) \wedge \Box(a_1 \vee b_1))$$

To verify the validity of the projection, we need to verify that $\Pi_X(C) \parallel \Pi_{\bar{X}}(C) \preceq C$ *holds.*

Let us start by verifying that the guarantees of the composition are more constrained than φ_G*:*

$$\Box(a = a_0) \wedge \Box(a_0 \vee b_0) \wedge \Box(b = b_1) \wedge \Box(a_1 \vee b_1) \Rightarrow \Box(a \vee b)$$

The equation above is not always true. Consider, for instance, the case in which, at time 0, $a = a_0 = \mathit{False}$*,* $b_0 = \mathit{True}$*,* $a_1 = \mathit{True}$*, and* $b = b_1 = \mathit{False}$*. Thus, the projection* $\Pi_X(C)$ *is not valid. If, instead, the guarantees were* $\psi = \Box(a \wedge b)$*, the set* $X = \{a\}$ *would yield a valid projection.*

The definition of valid projection justifies limiting the variable partition to the output variables. Let us assume that the set V in Definition 5.1 could include also input ports, i.e., a contract C and its projection $\Pi_V(C)$ shared both inputs and output variables in V. For all the input variables not included in V, similarly to what happens in Example 5.1, there would be some variables of $\Pi_V(C)$ not mapped to C. Conversely, $\Pi_{\bar{V}}(C)$ would have all the input variables in V not mapped to C. Thus, their composition $C' = \Pi_V(C) \parallel \Pi_{\bar{V}}(C)$ would have up to $2 \cdot |I|$ input variables. The refinement $C' \preceq C$ would then be problematic for two reasons, one merely formal and one substantial.

First, according to the definition of refinement in Section 3.2.1, for the refinement to hold, C' cannot have more input variables than C. C', however, would have twice the input variables of C. Although problematic, this issue could be resolved by adding up to $|I|$ new dummy variables to C, mapping them to the unconnected variables of C'. The other problem, however, is substantial. For each new input port, in fact, C' defines some constraints according to

[4]The new guarantees are computed according to the contract connection discussed in Section 3.2.1.

its assumptions, derived from $\Pi_V(C)$ and $\Pi_{\bar{V}}(C)$. Even if C had the missing variables, it would not have any assumption asserted over them, i.e., any behavior would be allowed. Thus, in verifying refinement, Equation 14c would not be satisfied as φ would allow more behaviors than φ', at least for any non-trivial assumption, i.e., different from $\varphi = \mathit{True}$.

With the notion of projection for an LTL A/G contract that we just introduced, we have all we need to state the problem we want to solve.

Definition 5.2 (Contract Decomposition Problem (CD))**.** *Let $C = (I, O, \varphi, \psi)$ be an LTL A/G contract, and let $\Pi_V(C)$ indicate the projection of C over a set of output variables $V \subseteq O$. The CD problem consists in partitioning O into n sets of variables $V_1, \ldots, V_n$, with $n \geq 1$, such that*

$$\Pi_{V_1}(C) \parallel \Pi_{V_2}(C) \parallel \cdots \parallel \Pi_{V_n}(C) \preceq C \tag{49}$$

5.2. Addressing the Contract Decomposition Problem

The first thing to do to solve the CD problem is to understand how to partition the set of output ports of a certain contract C to guarantee that Equation 49 is satisfied.

To get there, however, we first need to introduce a few new concepts. We will start defining the notion of independent variables for an LTL formula.

Definition 5.3 (Independent Variables)**.** *Let φ be an LTL formula over a set of variables Σ, and $\Sigma = Q \cup P$, with $Q \cap P = \emptyset$. Let also σ indicate a sequence of evaluations of the variables in Σ, where σ_P refers to the same sequence only over evaluations of variables in the set P. We say that variables in $V \subseteq P$ are independent in P for φ if and only if, for each sequence σ that falsifies φ, then φ can also be falsified only by the sequence $\sigma_{Q \cup V}$ of evaluations of variables in $Q \cup V$ or the sequence $\sigma_{Q \cup \bar{V}}$ of evaluations of variables in $Q \cup \bar{V}$, where $\bar{V} = P \setminus V$:*

$$\forall \sigma : \sigma \not\models \varphi \Rightarrow \sigma_{Q \cup V} \not\models \varphi \vee \sigma_{Q \cup \bar{V}} \not\models \varphi$$

Independent variables are useful because they allow identifying partitions of variables that can indicate the failure of a larger formula. They will be used as the foundational concept to identify valid contract projections.

Example 5.2. *Let $\varphi = \square(a \vee b)$, where $\Sigma = P = \{a, b\}$, and $Q = \emptyset$. Then $V = \{a\}$ does not contain independent variables*[5]*. Consider, for instance, the finite sequence $\sigma = [(a = \mathit{False}, b = \mathit{False})]$. σ falsifies φ, but $\sigma_V = [(a = \mathit{False})]$ does not, because the evaluation of b is unknown. On the other hand, for $\varphi' = \square(a \wedge b)$ and $V = \{a\}$, a is independent. Note that, trivially, variables in $V' = \{a, b\}$ are also independent.*

The following theorem is very useful as it defines the link between valid projections and independent variables for a certain formula, which is at the core of the algorithms that we will describe in Section 5.3.

[5]If the context is clear, as in this case, we will just say that variables in V are independent.

Theorem 5.1. *Let $C = (I, O, \varphi, \psi)$ be a compatible and consistent LTL A/G contract, and consider a subset of its output variables $V \subseteq O$. Let also φ be defined only over variables in I. Variables in V are independent in O for ψ if and only if the projection $\Pi_V(C)$ is valid.*

Proof. We start by proving that if variables in V are independent, then $\Pi_V(C)$ is valid. Consider a contract $C = (I, O, \varphi, \psi)$ and a subset of variables $V \subseteq O$, where V contains independent variables. To verify the validity of the projection $\Pi_V(C)$, then $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ must hold. This means verifying that:

1. $\varphi \rightarrow \varphi_V \wedge \varphi_{\bar{V}} \vee \neg(\psi_V \wedge \psi_{\bar{V}})$
2. $\psi_V \wedge \psi_{\bar{V}} \rightarrow \psi$

Let us begin with point 2). C is consistent, hence I-receptive, i.e., its guarantees cannot be falsified only by a sequence of evaluations of its input variables. Let σ_G be a sequence of evaluations of the variables of C that falsifies ψ. Then, because V contains independent variables, the same sequence projected over $I \cup V$ and $I \cup \bar{V}$ will also falsify ψ. Thus, ψ_V or $\psi_{\bar{V}}$ will also be false as they share the variables in $I \cup V$ and $I \cup \bar{V}$ with ψ, respectively. Hence, the implication in point 2) will always be true, as every time the right-hand side is false, so is the left-hand side.

For point 1), let σ_A be a sequence of evaluations of variables in C that falsifies the formula on the right-hand side of the implication, $\varphi_V \wedge \varphi_{\bar{V}} \vee \neg(\psi_V \wedge \psi_{\bar{V}})$, which represent the assumptions of the composition $\Pi_V(C) \parallel \Pi_{\bar{V}}(C)$. This means that σ_A falsifies also the stronger formula $\varphi_V \wedge \varphi_{\bar{V}}$. As C, $\Pi_V(C)$, and $\Pi_{\bar{V}}(C)$ all share the same set of input variables I, and φ is defined only over those variables, any σ_A that falsifies $\varphi_V \wedge \varphi_{\bar{V}}$ will also falsify φ. Hence, 1) is always true, too.

Now we need to prove that if the projection $\Pi_V(C)$ is valid, then the variables in V are independent in O for ψ. If $\Pi_V(C)$ is valid, then points 1) and 2) hold. Let us assume that the variables in V are not independent. Then, considering point 2), there should be a sequence σ_G that falsifies ψ but not ψ_V or $\psi_{\bar{V}}$. But that would mean that point 2) does not hold, as we can falsify the right-hand side of the implication but not the left-hand side, which contradicts $\Pi_V(C)$ being valid. That means that V must contain independent variables.

This proves the theorem. □

5.2.1. Using Projections for Synthesis

At this point, we know how to identify valid projections for a contract. In this section, we describe how such projections can be used to simplify the problem of synthesis from contract libraries.

We still have to make sure that given some projections of a contract C, we can indeed compose them together such that their composition is a refinement of C, as indicated in Equation 49. The following theorem clarifies this point.

Theorem 5.2. *Let $\Pi_{V_1}(C), \ldots, \Pi_{V_n}(C)$ be valid projections of a contract $C = (I, O, \varphi, \psi)$, which is compatible, consistent, and φ is defined only over variables in I. If $V_1, \ldots, V_n$ all contain variables independent in O for ψ, and $V_1 \cup \cdots \cup V_n = O$, then*

$$\Pi_{V_1}(C) \parallel \cdots \parallel \Pi_{V_n}(C) \preceq C \tag{50}$$

Proof. By the definition of valid projection in Equation 48, we know that for each projection $\Pi_{V_i}(C)$ of a contract (I, O, φ, ψ), the refinement $\Pi_{V_i}(C) \parallel \Pi_{\bar{V}_i}(C) \preceq C$ holds, where $V_i \cup \bar{V}_i = O$ by construction. To prove Equation 50 we need to verify that:

1. $\varphi \rightarrow \varphi_{V_1} \wedge \cdots \wedge \varphi_{V_n} \vee \neg(\psi_{V_1} \wedge \cdots \wedge \psi_{Vn})$
2. $\psi_{V_1} \wedge \cdots \wedge \psi_{V_n} \rightarrow \psi$

The proof, at this point, is similar to the proof of Theorem 5.1. We start from point 2). For σ_G being a sequence of evaluations of variables of C which falsifies ψ, we know that there must exist a $V_x \subseteq O$ with independent variables such that the guarantee ψ_{Vx} of the projection $\Pi_{Vx}(C)$ is false, too. Without loss of generality, let us assume that V_x is also minimal, i.e., there not exists a proper subset of V_x which also contains independent variables. Since $\{V_1, \ldots, V_n\}$ all contain independent variables and $V_1 \cup \cdots \cup V_n = O$, then there exists a $V_i \in \{V_1, \ldots, V_n\}$ such that $V_x \subseteq V_i$. Thus, ψ_{V_i} will be falsified by σ_G, and so will be also the whole left-hand side of the implication at point 2). This proves that point 2) always holds. The proof of point 1) is exactly the same as the one of point 1) in Theorem 5.1. Hence, the theorem is proved. □

Theorem 5.2 explains how we can partition a contract C using n valid projections, $\Pi_{V_1}(C), \ldots, \Pi_{V_n}(C)$. This is good news because now we can synthesize a composition of contracts that refines C from a library L, if such a composition exists, by independently synthesizing compositions for the projections $\Pi_{V_1}(C), \ldots, \Pi_{V_n}(C)$. That is, if there exist compositions such that

$$C_1^{V_i} \parallel \cdots \parallel C_{m_i}^{V_i} \preceq \Pi_{V_i}(C), \qquad 1 \leq i \leq n$$

for some $m_1, \ldots, m_n$, then by the *independent development* property in [17], the following holds:

$$\left.\begin{array}{c} C_1^{V1} \parallel \cdots \parallel C_{m1}^{V1} \\ \parallel \\ C_1^{V2} \parallel \cdots \parallel C_{m2}^{V2} \\ \parallel \\ \vdots \\ \parallel \\ C_1^{Vn} \parallel \cdots \parallel C_{mn}^{Vn} \end{array}\right\} \preceq \left.\begin{array}{c} \Pi_{V1}(C) \\ \parallel \\ \Pi_{V2}(C) \\ \parallel \\ \vdots \\ \parallel \\ \Pi_{Vn}(C) \end{array}\right\} \preceq C \tag{51}$$

However, we are not done yet. Each projection $\Pi_{V_i}(C)$, in fact, has even more variables than C (the copies of variables of C plus those added through the interconnect θ), although now we are only interested in the subset V_i. For the other variables, in fact, we know that there will be another composition taking care of them. Thus, we need a way to limit synthesis only to V_i. We address this issue by defining, and solving, a variant of the CSCL problem in Definition 4.1.

Definition 5.4 (Problem of Partial Synthesis from LTL A/G Contract Libraries (PCSCL))**.** *Let the LTL A/G contract $S = (I_S, O_S, \varphi_S, \psi_S)$ be a system specification, V be a subset of independent variables $V \subseteq O_S$, and define $\bar{V} = O_S \setminus V$. Let also $\Pi_{\bar{V}}(S)$ be the projection of S over $\bar{V}$, and $L = (\mathcal{Z}, \mathcal{R})$ be a library where $\mathcal{R}$ specifies that no further connection is required for variables of S in $\bar{V}$ and*

variables of $\Pi_{\bar{V}}(S)$*. Then, find a set of contracts* $H = \{C_1, \ldots, C_m\} \subseteq \mathcal{Z}$*, and an interconnect* $\theta \subseteq \mathcal{R}$ *such that the following holds:*

$$(C_1 \parallel \cdots \parallel C_m)\theta \parallel \Pi_{\bar{V}}(S) \preceq S$$

To solve this synthesis problem, the synthesizer is forced to use $\Pi_{\bar{V}}(S)$ as a placeholder connected to ports in $\bar{V}$, therefore it avoids wasting time in satisfying constraints for ports that are not in V. Overall, when instantiated for n subsets of independent variables $V_1, \ldots, V_n$, this results in n smaller synthesis problems which can be run independently, i.e., concurrently, using techniques such as the one proposed in Section 4.

Once we have found a solution for all the projections, Equation 51 guarantees that putting all the pieces back together will result in a proper refinement of our original system specification.

By only considering the variables in V we are also guaranteed that, with this decomposition approach, we are not reducing the solution space of the full synthesis problem. That is, the projection $\Pi_V(C) \parallel \Pi_{\bar{V}}(C)$ refines C, but it only restricts its guarantee over variables that are not mapped to C. Indeed, by definition of independent variables, each falsifying trace σ will falsify $\Pi_V(C)$ either over V or over $\bar{V}$. If σ falsifies $\Pi_V(C)$ over variables in V, then, by construction, it will also falsify C. Hence, the solution space over V is the same for both C and $\Pi_V(C)$.

5.3. An Efficient Decomposition Algorithm

In this section, we discuss an efficient algorithm to decompose a contract $C = (I, O, \varphi, \psi)$ leveraging the concepts discussed in the previous sections. One obvious potential approach is to exhaustively verify whether $\Pi_{V_i}(C) \parallel \Pi_{\bar{V}_i}(C) \preceq C$ holds for all the possible $V_i \in \wp(O)$, where $\wp(O)$ is the powerset of O. This is not very efficient, as it requires checking Equation 48 at least $2^{|O|}$ times. We propose a better solution that only needs a quadratic number of checks.

The intuition is to start from sets V_i that contain single output variables of C and to use a model checker to suggest how to increase the size of each V_i until it contains independent variables. We do so by analyzing the counterexamples obtained by verifying some *ad hoc* formulas.

Algorithm 5 describes the main decomposition algorithm. For each output variable p (Line 3), we initialize the set V by including only that variable (Line 4). After creating the candidate projections $\Pi_V(C)$ and $\Pi_{\bar{V}}(C)$, the algorithm verifies that $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ in Line 8 (by model checking the refinement formulas). If $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ holds, then `passed` will be true, meaning that $\Pi_V(C)$ is a valid projection (hence, V contains independent variables) and we can move on to the next iteration (Line 14). If, however, $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ can be falsified, the model checker will generate a counterexample that proves why the refinement does not hold. We can then analyze the counterexample to identify which variables were responsible for the failure, i.e., behaved differently in the two projections, and add them to V (Lines 10 and 11), repeating the process until $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ is valid. Finally, the last step of `DecomposeContract`, Line 16, guarantees that the set $\{V_1, \ldots, V_n\}$ is a partition of O, as required by Definition 5.2. That is, `MergeClusters` will merge any two sets in $\{V_1, \ldots, V_n\}$ that have variables in common, resulting in a new set $\mathsf{clusters} = \{V_1, \ldots, V_m\}$ such that $\forall V_i, V_j \in \{V_1, \ldots, V_m\} : V_i \cap V_j = \emptyset$. The algorithm always terminates,

```
1  function DecomposeContract:
       Input: Contract C = (I, O, φ, ψ)
       Output: Set of valid contract projections
2      clusters ← {};
3      for p ∈ O do
4          V ← {p};
5          passed ← False;
6          repeat
7              V̄ ← O \ V;
8              passed, trace ← checkValid(Π_V(C) ∥ Π_V̄(C) ⪯ C);
9              if not passed then
10                 D ← ParseTrace(trace);
11                 V ← V∪D;
12             end
13         until passed;
14         clusters ← clusters ∪ V;
15     end
16     clusters ← MergeClusters(clusters);
17     return {Π_V(C) | V ∈ clusters }
18 end
```

Algorithm 5: Contract Decomposition algorithm. It takes a contract C as input, and returns a set of n projections such that $\Pi_{V_1}(C) \parallel \cdots \parallel \Pi_{V_n}(C) \preceq C$.

as in the worst case we have that $V = O$, thus $\Pi_V(C) = C$ and $\bar{V} = \emptyset$, which always verifies $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$. It does so by invoking the model checker (through the function `checkValid`, which solves a PSPACE-complete problem, although very small if compared to the CSCL synthesis framework) at most n^2 times, where n is the number of output ports of C.

5.3.1. Algorithm `DecomposeContract` is sound

In this section, we show that Algorithm `DecomposeContract` is sound, meaning that the contract projections that it returns are always valid. To do so, we need to show that at the end each V contains independent variables. We always start from a set V containing a single variable. In the main iteration, for each candidate V, the function `checkValid` in `DecomposeContract`, Line 8, returns true if $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ is true, which means that the projection over V is valid. We say so because, otherwise, the model checker would have found a trace that violates C's guarantees but is not accepted by $\Pi_V(C) \parallel \Pi_{\bar{V}}(C)$, according to Definition 5.3. By Theorem 5.1, then V contains independent variables. If the refinement does not hold, then it means that the variables in V are not independent, that is, V is too small. Therefore, the variables in V depend on (at least) a variable in $\bar{V}$. The model checker provides a counterexample that shows which variables caused the refinement to fail, i.e., they behave differently between the two projections. Among these variables, some must be dependent on variables on V, because, otherwise, the refinement would hold. By parsing the counterexample trace we can recognize such variables

and add them to V, repeating the verification step in Line 8. This process stops when $\Pi_V(C) \parallel \Pi_{\bar{V}}(C) \preceq C$ is true, which means, as we have seen above, that V contains independent variables. To make sure we capture all the possible sets of independent variables, we repeat the process starting with every variable in O. In the worst case, the process stops when $V = O$, which will always result in a valid projection. Finally, `MergeClusters` will take care of resolving any ambiguity by conservatively merging any two sets V_i, V_j in case they have any variables in common. This can happen because the variable sets inferred by counterexample traces are not always minimal, but it does not affect the correctness of the algorithm. The algorithm is therefore sound.

We cannot claim that the algorithm is complete, meaning that if there exists a partition $V_1, \ldots, V_n$ such that $\Pi_{V_1}(C) \parallel \cdots \parallel \Pi_{V_n}(C) \preceq C$, with $n > 1$, we cannot guarantee that the algorithm will find it. The reason is that by analyzing the counterexample we cannot be sure that some variables that are different between the two contracts are indeed part of the cause of the failed refinement check. In practice, however, we observed that the model checker we used, nuXmv, tends to show counterexamples in which only the variables effectively responsible for the refinement to fail are different between the two projections.

6. Case Studies

To evaluate the techniques we presented, here we introduce three industry-relevant case studies. These are interesting problems because they are simple enough to allow a non-domain expert reader to follow them, yet, they highlight challenges and the capabilities of our approach. We implemented the algorithm in Python, as an extension of the tool Pyco, introduced in [21], that uses the nuXmv model checker to compute the validity and satisfiability of formulas, and to execute the state machines generating the counterexample traces. For each case study, we evaluated the performance of the proposed synthesis techniques (Algorithm 4) with and without the application of the Specification Decomposition (SD) strategy (Algorithm 5). To ensure accurate reporting, we ran each synthesis task several times (30 to 80 times) and then bootstrapped the samples to compute the mean of the synthesis time and the related 95% confidence intervals. The experiments in Sections 6.1 and 6.2 were executed on a 2.5 GHz Intel Core i7 machine with 16GB of RAM; the experiments in Section 6.3, instead, were executed on a 3.3 GHz Intel Xeon machine with 32 GB of RAM.

6.1. The SPI Analog-to-Digital Converter problem

An Analog-to-Digital Converter (ADC) is an electronic device that is able to read an analog voltage and returns a digital value proportional to the input voltage. ADCs are key in the design of cyber-physical systems, as they provide systems with the ability to "read" information that is generated by interacting with the physical world. For instance, consider a simple system that needs to detect when the environment in which it operates is dark. An immediate implementation could include a microcontroller (MCU) and a photoresistor, which is a sensor able to convert light to electrical potential. The MCU, however, is a digital component and needs an ADC to interpret the output of the sensor. The communication between the MCU and the ADC, usually, happens through

a serial digital signal. A typical protocol used in this type of communication is called Serial Peripheral Interface (SPI) (see, for instance, [71]).

In this example, we focus on the interaction between components implementing an SPI-like serial protocol, and a specification requiring all the information in parallel instead. The specification models a subsystem for an embedded device, and requires reading from an analog source and returning its corresponding digital value using a bus, i.e., in parallel. Here, the specification does not implement a serial protocol. It asserts a signal, r, for a single clock cycle and expects the digital value to be produced on the bus after some time. Table 2 provides the details about the contract representing the specification.

Input Ports	a r	(Analog:Int) (Req:Bool)
Output Ports	rdy $b_0, \ldots, b_n$	(Ready:Bool) (ADCbit:Bool)
Assumptions	$\neg r \wedge \square(r \rightarrow \bigcirc\neg r \wedge \cdots \wedge \bigcirc^{n+2}\neg r) \wedge \square(0 \leq a < 2^n)$	
Guarantees	$\square(r \rightarrow \bigcirc\neg rdy \wedge \cdots \wedge \bigcirc^{n+1}\neg rdy \wedge \bigcirc^{n+2} rdy) \wedge$ $\square\{r \rightarrow [(\bigcirc^{n+2} b_0 \leftrightarrow \bigcirc^2(\frac{a}{2^0} - 2 \cdot \frac{a}{2^1} = 1)) \wedge \cdots \wedge (\bigcirc^{n+2} b_n \leftrightarrow \bigcirc^2(\frac{a}{2^{n-1}} - 2 \cdot \frac{a}{2^n} = 1))]\}$	

Table 2: Specification for the SPI-ADC synthesis problem parameterized for n (in the experiments, $n \in [2, 8]$). The contract reads an analog input, represented as an integer variable, and a request line. After $n + 2$ clock cycles, it needs to assert a ready signal. Moreover, it needs to compute the i-th bit of the analog input for each output bit b_i, representing the state of the analog signal two cycles after the request line was asserted. The fractions represent integer division, where the expression $\frac{a}{2^i} - 2 \cdot \frac{a}{2^{i+1}}$ computes the i-th bit of a. Port types indicate the type label, used to describe each port, and their domain (integer or Boolean).

The analog input is represented by an integer variable. To satisfy the specification, an implementation needs to match the input value to its digital binary version. The specification requires also a "ready" signal, rdy, to be generated upon completion of the task.

Table 3 shows, instead, the elements that populate the contract library. The set of contracts is straightforward. The main component is the ADC, which implements an SPI-like protocol: it requires its c input to be asserted for several clock cycles, while it samples from its analog interface and generates corresponding digital values on its serial output, m. Being serial means that the analog value will not be converted at once, but each bit will be computed sequentially. The rest of the library contains simpler digital elements, such as inverters, which reverse their Boolean input value, counters, which count up to a certain integer value, comparators, which return true if their input matches their parameter, triggers, which return a Boolean signal after a certain number of clock cycles, and flip-flops, which store their Boolean input value when their "write" line is asserted. Note that most of the elements in the library are parametric, with a parameter $0 \leq k \leq 10$, meaning that each of those contracts effectively implies a larger search space. Figure 1 illustrates the difference between the ADC in the library and the specification. The goal of this experiment is to synthesize the control logic around the ADC, meaning that a correct solution will need to figure out a way to memorize intermediate values and return the binary representation of the input only when all the bits have been correctly computed. This case study is particularly interesting because its complexity grows also in the temporal dimension. For different specifications, in fact, we have that having more output ports implies that the ADC will need to work over more clock cycles, meaning that the underlying model checker will need to

Contract	Input Ports	Output Ports	Assume	Guarantee
ADC(n)	c (Select:Bool) a (Analog:Int)	m (SerialOut:Bool)	$\neg c$	$\neg m \wedge \bigcirc\neg m \wedge$ $\square(\neg c \rightarrow \neg m \wedge \bigcirc\neg m) \wedge$ $\square\{(\neg c \wedge \bigcirc c \wedge \bigcirc^2 c) \rightarrow$ $[\bigcirc^2 m \leftrightarrow \bigcirc(\frac{a}{2^0} - 2 \cdot \frac{a}{2^1} = 1)]\} \wedge$ $\cdots$ $\square\{(\neg c \wedge \bigcirc c \wedge \cdots \wedge \bigcirc^{n+1} c) \rightarrow$ $[\bigcirc^{n+1} m \leftrightarrow \bigcirc(\frac{a}{2^{n-1}} - 2 \cdot \frac{a}{2^n} = 1)]\} \wedge$ $\square[(\neg c \wedge \bigcirc c \wedge \cdots \wedge \bigcirc^{n+1} c) \rightarrow (\bigcirc^{n+2} \neg m)] \wedge$ $\square[(c \wedge \bigcirc c \wedge \cdots \wedge \bigcirc^{n+1} c) \rightarrow (\bigcirc^{n+2} \neg m)]$
Inverter	i (Control:Bool)	o (Control:Bool)	true	$\square(o \leftrightarrow \neg i)$
Counter(k)	r (Control:Bool)	v (Counter:Int) p (Param:Int)	true	$(c = 0) \wedge$ $\square(p = k) \wedge$ $\square[r \rightarrow \bigcirc(v = 0)] \wedge$ $\square[(v \leq p) \wedge \neg r \rightarrow (\bigcirc(v) = v + 1)] \wedge$ $\square[(v = p) \wedge \neg r \rightarrow (\bigcirc(v) = p)]$
Comparator(k)	v (Counter:Bool)	e (Control:Int) p (Param:Int)	true	$\square(p = k) \wedge$ $\square[(v = p) \leftrightarrow e]$
Trigger(k)	r (Control:Bool)	t (Control:Bool) c (Counter:Int) p (Param:Int)	true	$(c = 0) \wedge$ $\square(p = k) \wedge$ $\square[(c = p) \leftrightarrow t] \wedge$ $\square[r \rightarrow \bigcirc(c = 0)] \wedge$ $\square[(c \leq p) \wedge \neg r \rightarrow (\bigcirc(c) = c + 1)] \wedge$ $\square[(c = p) \wedge \neg r \rightarrow (\bigcirc(c) = p)]$
FlipFlop	d (Data:Bool) e (Control:Int)	q (Data:Bool)	true	$\neg q \wedge$ $\square[e \rightarrow (d \leftrightarrow \bigcirc q)] \wedge$ $\square[\neg e \rightarrow (q \leftrightarrow \bigcirc q)]$

Table 3: Structure of the SPI-ADC library. The ADC component is used, in the experiments, with $n \in [2, 8]$. Some components are parameterized if, in their output ports, there are ports of type "Param". All the parameters k are bounded such that $0 \leq k \leq 10$. Port types indicate the type label, used to describe each port, and their domain (integer or Boolean). The library imposes constraints on possible connections based on the port domains.

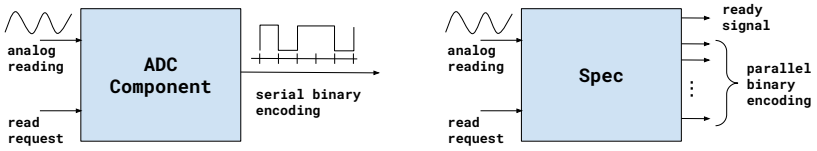


Figure 1: Comparison between the ADC and the problem specification. The ADC is able to sample an analog value and it returns a serial binary reading. The specification, however, requires a parallel binary reading.

do more work to verify candidates.

Resolution	Sample Mean, w/o SD	Sample Mean, w/ SD	Timeouts, w/ SD
2-bit	30.36 (28.96, 31.38)	28.91 (26.63, 31.75)	0
3-bit	46.04 (44.4, 47.69)	48.45 (44.62, 53.3)	0
4-bit	347.83 (323.66, 369.36)	106.16 (97.42, 115.5)	0
5-bit	6816.03 (-, -)	217.1 (161.59, 355.48)	2
6-bit	- (-, -)	299.42 (243.32, 409.69)	1
7-bit	- (-, -)	459.87 (361.62, 594.2)	6
8-bit	- (-, -)	614.54 (498.39, 737.5)	10

Table 4: Summary of the SPI synthesis experiments without and with SD, for specs requiring 2- to 8-bit resolution. Synthesis tasks were set to timeout at 1000 seconds, except the 5-bit experiment without SD, which was a one-time test. We report the bootstrap mean synthesis time and, in parentheses, its 95% confidence interval. Mean values are expressed in seconds. We also report the number of timeouts we observed for each problem with SD. The sample means we report include these timeouts. No timeouts were observed in the experiments without SD for 2- to 4-bit ADC resolution.

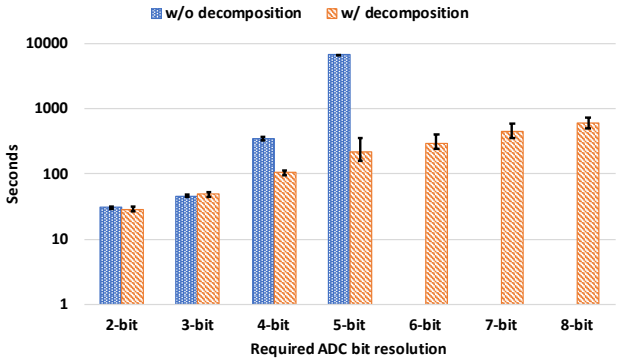


Figure 2: Comparison between the mean time required to synthesize a solution to the SPI-problem with and without SD. Each bar represents the bootstrap mean for the synthesis time, while the dark error bars represent its 95% confidence interval. Synthesis was set to timeout at 1000 seconds. For the 5-bit case without SD, we are reporting the result of a single experiment. The graph is in logarithmic scale.

We ran 80 experiments without SD for each specification requiring 2-, 3-, and 4-bit ADC resolution, and libraries with 11, 11, and 16 contracts, respectively. We also ran a one-off test with a specification requiring 5-bit resolution and no SD, with a library of 17 contracts. The experiment was not repeated as it took over two hours to finish, but we still report it as it shows how synthesis performance can degrade quickly. When testing with SD, we ran 30 experiments for each specification requiring 2- to 8-bit ADC resolution. For these experiments, we use libraries with 14 elements. Each library had an appropriate ADC contract, according to the specification being tested, while the other contracts, including the parametric ones, were replicated as needed.

Figure 2 illustrates the results of our experiments. In the figure, each bar represents the bootstrap mean for each specification, while the error bars indicate their 95% bootstrap confidence intervals. In the experiments without SD, we observe how the synthesizer can find solutions in a reasonable time (less than 1000 seconds) for $n < 5$. For $n \geq 5$, synthesis for this problem becomes unpractical. When applying SD, we can synthesize more complex problems in less than 1000 seconds. Table 4 summarizes the same experiments showing the values of the bootstrap mean and 95% confidence interval for all the experiments. It also shows how many times, for each specification, synthesis was aborted because it took too long (>1000 seconds). We only observed timeouts for $n >= 5$ in the experiments with SD. While we have not seen timeouts for lower resolutions, one-third of the experiments for the 8-bit specification timed out. In our opinion, this behavior is not surprising and is consistent with the expectation that more complex specifications have higher synthesis time variability.

The benefit of decomposing the specification, for this case study, can be observed immediately: thanks to that technique, we can solve problems that otherwise would be practically unfeasible. Figure 3 shows the typical solution for a specification requiring an ADC with 8-bit resolution. As one can see, the synthesizer correctly inferred the parameters for several components (e.g., the triggers), to make sure that the rest of the signals had the correct timing. Here, parameters are just a special type of output variable. We believe that this case

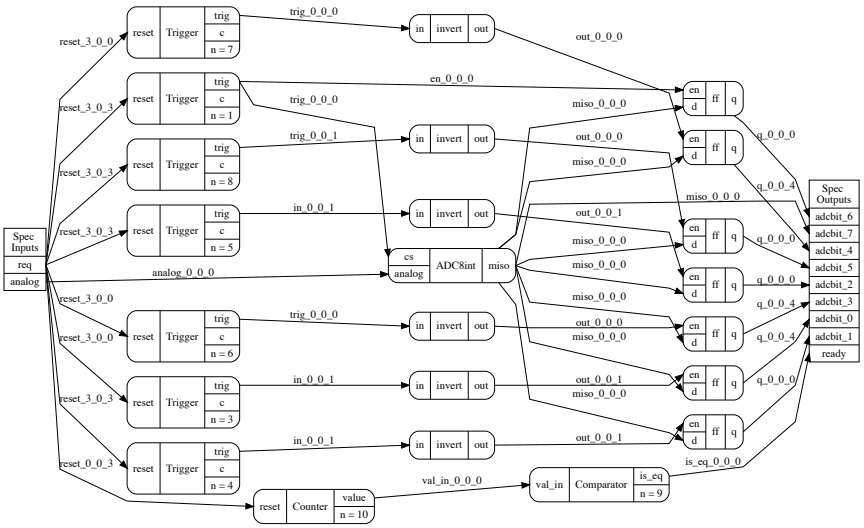


Figure 3: Typical synthesized composition for the SPI-ADC problem, in the case of the specification and ADC with 8-bit resolution. Triggers, Counter, and Comparator in the picture refer to a certain value n. That value is the representation of the parameter k listed the library.

study greatly benefited from the application of SD because of the particular structure of the specifications, where we had several outputs that needed to be satisfied at once. Indeed, the decomposition algorithm was able to show that each output variable of the specification was independent of the rest, simplifying the synthesis problem. We could not evaluate this problem with the tool presented in [21] because here specifications and library use primitives, such as integers, that are not supported in that release.

6.2. The BLDC problem

The goal of this design example is to synthesize the architecture of a controller for a brushless DC electric motor, starting from components such as a microcontroller or DC-DC converters. This example was first introduced and evaluated in [21]. In [21], the library of contracts for this problem (Table 5)

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

Table 5: 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.

Contract	Input Ports	Output Ports	Assumptions	Guarantees
Power-5V	- -	*gnd* (GND) *vout* (Voltage5V)	true	$\square(\neg gnd \wedge vout)$
DCDC-3V	*gnd* (GND) *vin* (Voltage12V)	*vout* (Voltage3V)	true	$\square(vout)$
DCDC-5V	*gnd* (GND) *vin* (Voltage12V)	*vout* (Voltage5V)	true	$\square(vout)$
Power-12V	- -	*gnd* (GND) *vout* (Voltage12V)	true	$\square(\neg gnd \wedge vout)$
MCU	*gnd* (GND) *vin* (Voltage3V) *i* (IOPin3V)	o_1 (IOPin3V) o_2 (IOPin3V) o_3 (IOPin3V)	true	$o_1 \wedge \neg o_2 \wedge \neg o_3 \quad \wedge$ $\square[\ (o_1 \wedge \neg i \wedge \bigcirc i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[(o_1 \wedge \neg i \wedge \bigcirc\neg i) \rightarrow (\bigcirc o_1 \wedge \bigcirc\neg o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[\ (o_1 \wedge i \wedge \bigcirc\neg i) \rightarrow (\bigcirc o_1 \wedge \bigcirc\neg o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[\ (o_2 \wedge \neg i \wedge \bigcirc i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc o_2 \wedge \bigcirc o_3)\ ] \wedge$ $\square[(o_2 \wedge \neg i \wedge \bigcirc\neg i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[\ (o_2 \wedge i \wedge \bigcirc\neg i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[\ (o_3 \wedge \neg i \wedge \bigcirc i) \rightarrow (\bigcirc o_1 \wedge \bigcirc\neg o_2 \wedge \bigcirc\neg o_3)\ ] \wedge$ $\square[(o_3 \wedge \neg i \wedge \bigcirc\neg i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc\neg o_2 \wedge \bigcirc o_3)\ ] \wedge$ $\square[\ (o_3 \wedge i \wedge \bigcirc\neg i) \rightarrow (\bigcirc\neg o_1 \wedge \bigcirc\neg o_2 \wedge \bigcirc o_3)\ ] \wedge$ $\square\{\quad (i \leftrightarrow \bigcirc i) \quad \rightarrow [\quad (\bigcirc o_1 \leftrightarrow o_1) \wedge$ $(\bigcirc o_2 \leftrightarrow o_2) \wedge (\bigcirc o_3 \leftrightarrow o_3)]\}$
Half-Bridge	*gnd* (GND) *vin* (Voltage3V) *i* (IOPin3V)	*o* (IOPin12V)	true	$\square(i \leftrightarrow o)$

Table 6: Structure of the BLDC library, which contains three separate instances of each component. This library is a revisited version of the one in Table 5 of [21], which now includes only deterministic contracts.

contains contracts that are nondeterministic, which can lead to performance issues as discussed in Section 4.3.1. Specifically, the contract representing the microcontroller, MCU, is a problematic one. For instance, here the contract does not specify what to do in case the input i is true for more than one cycle. Table 6 shows an updated library with only deterministic contracts. The identification of the nondeterministic contracts and the validation of the updated library was performed using Algorithm 3. Table 5 shows the requirements of the overall system that we want to synthesize. This table is taken verbatim from Table 4 of [21]. We considered three different libraries based on Table 6 with 16, 24, and 32 contracts, and used them to synthesize the same specification, with and without applying the SD techniques. For each category, we ran 80 experiments.

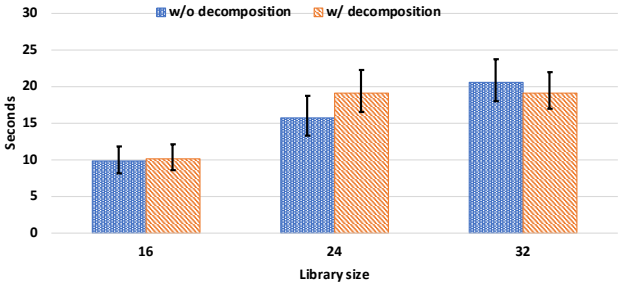


Figure 4: Summary of the results for the BLDC experiments. We synthesized a controller using three libraries including 16, 24, and 32 contracts respectively, for the same specification, without and with SD. Each bar represents the bootstrap mean synthesis time, while the error bars represent the 95% confidence intervals.

Figure 4 shows the results for all the experiments. Each bar represents the bootstrap mean synthesis time while the error bars indicate the 95% bootstrap confidence interval for each mean. Table 7 summarizes the same results. In these experiments, the solver was able to find a solution almost immediately—most

Library Size	Sample Mean, w/o SD	Sample Mean, w/ SD
16	9.87 (8.22, 11.77)	10.14 (8.57, 12.09)
24	15.74 (13.37, 18.77)	19.18 (16.59, 22.25)
32	20.61 (17.96, 23.81)	19.18 (17.04, 21.97)

Table 7: Summary of the BLDC synthesis experiments, with libraries including 16, 24, and 32 contracts, respectively, with and without SD. We report the bootstrap mean synthesis time and, in parentheses, its 95% confidence interval. All values are expressed in seconds.

of the time, it took only two iterations to find a good solution. Synthesis using the nondeterministic library generated comparable results, although with much higher variance. For instance, for a certain library, it would not be uncommon to see some experiments taking more than five times the synthesis time of others.

We see that the synthesizer performance when using SD is comparable, if not slightly worse, than without it. In this case, the overhead of the application of the SD algorithm is greater than the benefit it provides. Yet, the fact that the performance cost is negligible is still useful. It suggests, in fact, that there is no harm in decomposing a specification even for a problem that can already be efficiently solved. Synthesis, finally, was significantly faster than in [21]. There, this example was synthesized only using a library with 16 elements. With the same library size, even if we used a less powerful CPU, here we are 20% faster when compared to the best case (minimal number of components in the solution), and 10 times faster when compared to the general constant-cost case. The resulting compositions were similar to the one in Figure 7 of [21].

6.3. The EPS problem

The goal of this example is to synthesize the architecture of a Bus Power Control Unit, from a set of subsystem controllers, for an aircraft Electrical Power System (EPS). The function of the controller is to react to changes in system conditions or failures and reroute power by actuating the contactors, ensuring that essential buses are adequately powered. This example was first introduced and evaluated in [31] and [21].

Table 8 illustrates the set of specifications that the BPCU needs to satisfy. The goal is to control the EPS represented in Figure 5. Table 9, instead, shows

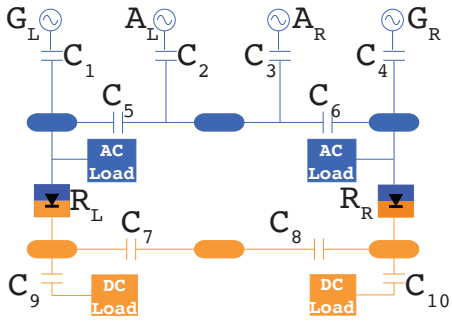


Figure 5: Simplified single line diagram of the EPS [72].

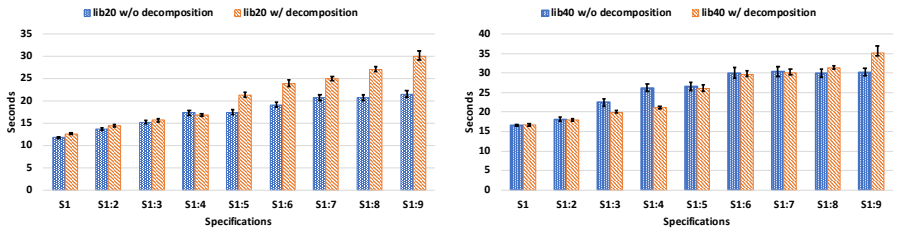
the contracts in the EPS library as defined in [21]. Every component is described by its I/O ports (annotated with their types), and its specification as an A/G pair. All the components make some assumptions over the state of a certain type

Input Ports	G_L, G_R (ActiveGenerator) A_L, A_R (BackupGenerator) R_L, R_R (Rectifier)	$\boldsymbol{S_1}$	$C_1 \wedge \square(G_L \rightarrow \bigcirc\neg C_1)$
		$\boldsymbol{S_2}$	$C_4 \wedge \square(G_R \rightarrow \bigcirc\neg C_4)$
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)	$\boldsymbol{S_3}$	$\square(A_L \rightarrow \bigcirc\neg C_2)$
		$\boldsymbol{S_4}$	$\square(A_R \rightarrow \bigcirc\neg C_3)$
		$\boldsymbol{S_5}$	$\square\neg(C_2 \wedge C_3)$
		$\boldsymbol{S_6}$	$\square[(\neg G_L \wedge \neg G_R) \rightarrow \Diamond\neg(C_5 \wedge C_6)]$
Assumptions (common to all)	$\neg G_L \wedge \square(G_L \rightarrow \bigcirc G_L) \wedge$ $\neg G_R \wedge \square(G_R \rightarrow \bigcirc G_R) \wedge$ $\neg A_L \wedge \square(A_L \rightarrow \bigcirc A_L) \wedge$ $\neg A_R \wedge \square(A_R \rightarrow \bigcirc A_R) \wedge$ $\neg R_L \wedge \square(R_L \rightarrow \bigcirc R_L) \wedge$ $\neg R_R \wedge \square(R_R \rightarrow \bigcirc R_R)$	$\boldsymbol{S_7}$	$\square[(\neg G_L \wedge \neg A_L \wedge \neg A_R \wedge \neg G_R) \rightarrow \Diamond(\neg C_2 \wedge \neg C_3 \wedge \neg C_5 \wedge \neg C_6)]$
		$\boldsymbol{S_8}$	$\square[\neg(R_L \wedge R_R) \rightarrow C_9]$
		$\boldsymbol{S_9}$	$\square[\neg(R_L \wedge R_R) \rightarrow C_{10}]$

Table 8: 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.

of EPS component and provide a guarantee over the state of various 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 an active generator or a backup one), 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.

As for the BLDC case, also in this case a contract first defined in [21] was found to be nondeterministic using Algorithm 3. To ensure determinism, in the following experiments, we replaced contract A_1 of Table 9 with another instance of contract B_1.



(a) Library size 20

(b) Library size 40

Figure 6: Summary of the EPS experiments. We used two different libraries, one with 20 and the other with 40 contracts, and ran experiments with and without SD. Each bar represents the bootstrap mean synthesis time, while the error bars represent the 95% bootstrap confidence intervals.

For each specification set, we ran 30 experiments with SD and 80 without it. When decomposing the full specification, the decomposition algorithm generated 7 sets of independent variables: $V_1 = \{C_1\}, V_2 = \{C_4\}, V_3 = \{C_2, C_3, C_5, C_6\}, V_4 = \{C_7\}, V_5 = \{C_8\}, V_6 = \{C_9\}$ and $V_7 = \{C_{10}\}$. Figure 6 summarizes the results, which are reported in detail in Table 10. The synthesizer's performance when

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)	$\neg f \wedge$ $\square(f \rightarrow \bigcirc f)$	$c \wedge$ $\square(\bigcirc f \rightarrow \bigcirc \neg c) \wedge$ $\square(\neg f \rightarrow c)$
$\mathbf{E_1}$	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)	$\neg f_1 \wedge \neg f_2$	$\square c$

Table 9: 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.

Specs	Lib 20, w/o SD	Lib 20, w SD	Lib 40, w/o SD	Lib 40, w SD
$\{S_1\}$	11.82 (11.66, 11.99)	12.62 (12.44, 12.8)	16.57 (16.35, 16.77)	16.75 (16.49, 17.05)
$\{S_1, S_2\}$	13.76 (13.49, 14.02)	14.42 (14.09, 14.77)	18.14 (17.65, 18.68)	18.02 (17.77, 18.33)
$\{S_1, \ldots, S_3\}$	15.26 (14.93, 15.65)	15.66 (15.33, 16.03)	22.47 (21.55, 23.48)	19.97 (19.69, 20.38)
$\{S_1, \ldots, S_4\}$	17.32 (16.81, 17.92)	16.91 (16.6, 17.21)	26.13 (25.22, 27.15)	21.19 (20.87, 21.52)
$\{S_1, \ldots, S_5\}$	17.41 (16.89, 17.99)	21.32 (20.83, 21.95)	26.45 (25.46, 27.52)	26.03 (25.36, 26.97)
$\{S_1, \ldots, S_6\}$	19.12 (18.54, 19.74)	23.79 (23.16, 24.69)	30.01 (28.69, 31.5)	29.71 (29.02, 30.61)
$\{S_1, \ldots, S_7\}$	20.7 (20.09, 21.39)	24.96 (24.53, 25.5)	30.36 (29.19, 31.64)	30.12 (29.5, 30.97)
$\{S_1, \ldots, S_8\}$	20.7 (20.15, 21.3)	27.07 (26.57, 27.63)	29.87 (28.84, 31.05)	31.35 (30.94, 31.78)
$\{S_1, \ldots, S_9\}$	21.45 (20.8, 22.25)	29.97 (29.13, 31.13)	30.25 (29.21, 31.31)	35.2 (34.29, 36.87)

Table 10: Summary of the EPS experiments, for libraries with 20 and 40 contracts. For each specification subset (one for each row), we report the bootstrap mean synthesis time and, in parentheses, its 95% confidence interval. All values are expressed in seconds.

using SD is comparable to its performance without it. As for the BLDC case, we need to consider that the decomposition algorithm is executed before synthesis at

the beginning of each experiment, and it's noteworthy that there is no significant impact on the overall synthesis performance. Compared to the results reported in Table 8 of [21], the approach presented in this paper is roughly twice as fast in the constant-cost case. This is not a surprise, as here each algorithm iteration uses significantly more information than in the other case, i.e., previous wrong candidates *and* execution traces, and therefore faster convergence is expected in general. Interestingly, while for the EPS experiments in the other work it was reported that an experiment generated more than 100000 verification queries, here no experiment took more than 8 iterations to converge to a solution. Figure 11 of [21] illustrates a typical composition resulting from the synthesis procedure.

7. Conclusions

The work described in this paper is centered on the problem of synthesizing a design by composing LTL A/G contracts available in a library.

First, we studied an approach to synthesis based on the CEGIS paradigm. Differently than the solution discussed in [21], here we use both the discarded topology `and` the counterexamples generated by a verifier, i.e., a model checker, representing traces over the specification and library variables, to propose a more efficient synthesis strategy. In this case, indeed, the number of queries to the verifier is reduced by several orders of magnitude, although the complexity of each query increases, too. Overall, we believe that our approach here is generally better, for LTL specifications, as each algorithm iteration uses more information to determine a new candidate and performance is significantly better. Furthermore, the algorithm we presented includes a number of optimizations that are aimed at improving performance, including considerations on why to avoid dependency cycles and nondeterministic contracts.

Then, we presented a technique to increase the scalability of synthesis from LTL A/G contracts libraries. We defined the notions of contract decomposition and projection and described an efficient algorithm to perform such decomposition. We tested this decomposition approach to two of the synthesis problems analyzed in Section 4. While the performance is comparable for smaller synthesis tasks, our decomposition strategy allows designers to manage complexity and synthesize designs also when they are not feasible with other techniques.

We applied our synthesis algorithm to three case studies to show its effectiveness. Those experiments provided valuable insights into the synthesis performance and showed the capabilities of our approach.

Limitations. We believe that the techniques proposed in this paper will play a role in the realization of more general platform-based design methodologies and tools handling LTL specifications. There are, however, some limitations that are important to discuss, many shared with the broader field of formal methods. Here, we assume that the libraries of contracts are available before starting the synthesis process. In practice, coming up with those libraries for a certain domain is not trivial. As one can see from our case studies, specifying LTL contracts for interesting components is a difficult and error-prone task, and automation is a necessity. We provide an effective tool to test libraries for non-determinism, i.e., Algorithm 3, but more work is needed to fully support a library designer. We are also aware that our decision to not allow cycles in

the synthesized compositions might be restrictive for some classes of problems. We showed how, in practice, feedback loops could negatively affect performance, and preventing cycles addresses the issue. We consider the possibility of allowing cycles, under specific circumstances, as future research. Finally, while this work significantly improves performance from what was observed in [21], scalability remains a concern, and tricks that could make synthesis faster for a certain specification might be counterproductive for a different one. Moreover, we are not aware of techniques to precisely estimate how long synthesis will take.

Future Directions. The synthesis technique described in Section 4 is efficient, but we don't have yet a way to specify sophisticated heuristics to help the synthesizer converge faster. In [21] we introduce cost functions, but they are limited to costs proportional to the number of ports and components in a solution. We would like, instead, to specify cost functions that can understand better the quality of a solution, rather than just its size. This could be done in several ways. One would be applying the human-in-the-loop strategy, where a designer's input on a partial design is used to define better search strategies. On the other hand, we can develop techniques to automatically infer the quality of a partial solution. The notion of quotient for contracts [48], for instance, can be used to represent "what is missing" in a certain composition to satisfy the specification. Additionally, [73] introduces a metric for LTL that characterizes a formula based on the number of traces that satisfy it. This metric can be leveraged, together with quotient, to devise algorithms able to build a solution including components according to their likelihood of satisfying part of the specification.

The notion of specification decomposition described in Section 5 is promising, as it allows the synthesis of bigger designs, faster, and with smaller contract libraries. The algorithm to identify sets of independent variables that we introduced, however, is sound but not complete. Additional work could help develop a new version of the algorithm that is both sound and complete. Furthermore, the effectiveness of our decomposition technique depends on the number of independent variables in each projection. One way to maximize the decomposability of a contract would be to introduce a pre-processing step in which the contract is refined into another one with smaller sets of independent variables. This could result in a tool suggesting to a designer how to "tweak" a certain specification to maximize synthesis performance.

Acknowledgements

The inception of the brushless motor example was aided by conversations with Richard Lin and Rohit Ramesh. The work in this paper has been supported by the National Science Foundation (NSF) via the project "ExCAPE: Expeditions in Computer Augmented Program Engineering" (CCF-1139138) and Contract CPS Medium 1739816, by the Camozzi Group, IBM, and United Technologies Corporation (UTC) via the iCyPhy consortium, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] 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. doi:10.1109/JPROC.2006.890107.

[2] D. D. Walden, G. J. Roedler, K. Forsberg, R. D. Hamelin, T. M. Shortell (Eds.), Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition, Wiley, Hoboken, NJ, 2015.

[3] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi, Benefits and challenges for platform-based design, in: Proceedings of the 41st Annual Design Automation Conference, DAC '04, ACM, New York, NY, USA, 2004, pp. 409–414. doi:10.1145/996566.996684.

[4] M. Sgroi, Platform-based design methodologies for communication networks, Ph.D. thesis, University of California, Berkeley (2002).

[5] B. P. Douglass, Agile Systems Engineering, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[6] 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.

[7] L. Lamport, What good is temporal logic?, Information Processing 83, R. E. A. Mason, ed., Elsevier Publishers 83 (1983) 657–668.

[8] 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.

[9] A. Bhatia, L. E. Kavraki, M. Y. Vardi, Sampling-based motion planning with temporal goals, in: 2010 IEEE International Conference on Robotics and Automation, 2010, pp. 2689–2696. doi:10.1109/ROBOT.2010.5509503.

[10] G. E. Fainekos, A. Girard, H. Kress-Gazit, G. J. Pappas, Temporal logic motion planning for dynamic robots, Automatica 45 (2) (2009) 343–352. doi:10.1016/j.automatica.2008.08.008.

[11] A. Armando, R. Carbone, L. Compagna, LTL model checking for security protocols, in: 20th IEEE Computer Security Foundations Symposium (CSF'07), 2007, pp. 385–396. doi:10.1109/CSF.2007.24.

[12] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, in: E. A. Emerson, K. S. Namjoshi (Eds.), Verification, Model Checking, and Abstract Interpretation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 364–380.

[13] N. Ozay, U. Topcu, R. M. Murray, Distributed power allocation for vehicle management systems, in: 2011 50th IEEE Conference on Decision and Control and European Control Conference, 2011, pp. 4841–4848. doi:10.1109/CDC.2011.6161470.

[14] T. Wongpiromsarn, U. Topcu, R. Murray, Automatic synthesis of robust embedded control software (2010).

[15] 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.

[16] 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.

[17] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, C. Sofronis, Multiple viewpoint contract-based specification and design, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Formal Methods for Components and Objects, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 200–225.

[18] L. de Alfaro, T. A. Henzinger, Interface theories for component-based design, in: T. A. Henzinger, C. M. Kirsch (Eds.), Embedded Software, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 148–165.

[19] 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.

[20] I. Filippidis, Decomposing formal specifications into assume-guarantee contracts for hierarchical system design, Ph.D. thesis, California Institute of Technology (07 2018).

[21] A. Iannopollo, S. Tripakis, A. Sangiovanni-Vincentelli, Constrained synthesis from component libraries, Science of Computer Programming 171 (2019) 21–41.

[22] S. Jha, S. A. Seshia, A theory of formal synthesis via inductive learning, Acta Informatica 54 (7) (2017) 693–726. doi:10.1007/s00236-017-0294-5.

[23] 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.

[24] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, V. Saraswat, Combinatorial sketching for finite programs, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, Association for Computing Machinery, New York, NY, USA, 2006, pp. 404–415. doi:10.1145/1168857.1168907. URL https://doi.org/10.1145/1168857.1168907

[25] S. Jha, S. A. Seshia, Are there good mistakes? a theoretical analysis of cegis, in: 3rd Workshop on Synthesis (SYNT), 2014, pp. 84–99.

[26] A. Iannopollo, S. Tripakis, A. Sangiovanni-Vincentelli, Specification decomposition for synthesis from libraries of LTL assume/guarantee contracts, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 1574–1579. doi:10.23919/DATE.2018.8342266.

[27] A. Pinto, A. L. Sangiovanni Vincentelli, Csl4p: A contract specification language for platforms, Systems Engineering 20 (3) (2017) 220–234. doi:10.1002/sys.21386.

[28] O. Grumberg, D. E. Long, Model checking and modular verification, ACM Transactions on Programming Languages and Systems 16 (1991).

[29] J. M. Cobleigh, D. Giannakopoulou, C. S. Pasareanu, Learning assumptions for compositional verification, in: H. Garavel, J. Hatcliff (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 331–346.

[30] A. Gupta, K. McMillan, Z. Fu, Automated assumption generation for compositional verification, Formal Methods in System Design (2008) 285–301.

[31] 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.

[32] A. Cimatti, S. Tonetta, A property-based proof system for contract-based design, in: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, 2012, pp. 21–28. doi:10.1109/SEAA.2012.68.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] L. Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[46] T. A. Henzinger, S. Qadeer, S. K. Rajamani, Decomposing refinement proofs using assume-guarantee reasoning, in: Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design, ICCAD '00, IEEE Press, Piscataway, NJ, USA, 2000, pp. 245–253.

[47] E. Dallal, P. Tabuada, Decomposing controller synthesis for safety specifications, in: 2016 IEEE 55th Conference on Decision and Control (CDC), 2016, pp. 5720–5725. doi:10.1109/CDC.2016.7799148.

[48] I. Incer, A. Sangiovanni-Vincentelli, C.-W. Lin, E. Kang, Quotient for assume-guarantee contracts, in: Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE '18, IEEE Press, 2018, pp. 67–77.

[49] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, ACM, New York, NY, USA, 1989, pp. 179–190. doi:10.1145/75277.75293.

[50] R. Rosner, Modular synthesis of reactive systems, Ph.D. thesis, Weizmann Institute of Science (1992).

[51] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Comput. 35 (8) (1986) 677–691. doi:10.1109/TC.1986.1676819.

[52] G. J. Holzmann, Explicit-state model checking, in: E. M. Clarke, T. A. Henzinger, H. Veith, R. Bloem (Eds.), Handbook of Model Checking, Springer International Publishing, Cham, 2018, pp. 153–171. doi:10.1007/978-3-319-10575-8_5.

[53] E. Filiot, N. Jin, J.-F. Raskin, Antichains and compositional algorithms for LTL synthesis, Formal Methods in System Design 39 (3) (2011) 261–296. doi:10.1007/s10703-011-0115-3.

[54] P. Wolper, M. Y. Vardi, A. P. Sistla, Reasoning about infinite computation paths, in: Proceedings of the 24th Annual Symposium on Foundations of Computer Science, SFCS '83, IEEE Computer Society, Washington, DC, USA, 1983, pp. 185–194. doi:10.1109/SFCS.1983.51.

[55] P. Gastin, D. Oddoux, Fast LTL to Büchi automata translation, in: G. Berry, H. Comon, A. Finkel (Eds.), Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), Vol. 2102 of Lecture Notes in Computer Science, Springer, Paris, France, 2001, pp. 53–65.

[56] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, R. Seeber, Ratsy – a new requirements analysis tool with synthesis, in: T. Touili, B. Cook, P. Jackson (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 425–429.

[57] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, R. M. Murray, Control design for hybrid systems with TuLiP: The temporal logic planning toolbox, in: 2016 IEEE Conference on Control Applications (CCA), 2016, pp. 1030–1041. doi:10.1109/CCA.2016.7587949.

[58] E. Filiot, N. Jin, J.-F. Raskin, An antichain algorithm for LTL realizability, in: A. Bouajjani, O. Maler (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 263–277.

[59] I. Filippidis, R. M. Murray, Symbolic construction of gr(1) contracts for systems with full information, in: 2016 American Control Conference (ACC), 2016, pp. 782–789. doi:10.1109/ACC.2016.7525009.

[60] I. Incer, The algebra of contracts, Ph.D. thesis, EECS Department, University of California, Berkeley (May 2022).
URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-99.html

[61] B. Meyer, Applying "design by contract", Computer 25 (10) (1992) 40–51. doi:10.1109/2.161279.

[62] R. W. Floyd, Assigning meanings to programs, Proceedings of Symposium on Applied Mathematics 19 (1967) 19–32.

[63] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580. doi:10.1145/363235.363259.

[64] T. A. Henzinger, S. Qadeer, S. K. Rajamani, You assume, we guarantee: Methodology and case studies, in: Proceedings of the 10th International Conference on Computer Aided Verification, CAV '98, Springer-Verlag, London, UK, UK, 1998, pp. 440–451.
URL http://dl.acm.org/citation.cfm?id=647767.733780

[65] 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.

[66] L. Doyen, T. A. Henzinger, B. Jobstmann, T. Petrov, Interface theories with component reuse, in: Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT '08, ACM, New York, NY, USA, 2008, pp. 79–88. doi:10.1145/1450058.1450070.

[67] E. A. Lee, A. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17 (12) (1998) 1217–1229. doi:10.1109/43.736561.

[68] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, I. Stierand, Using contract-based component specifications for virtual integration testing and architecture design, in: 2011 Design, Automation & Test in Europe, IEEE, 2011, pp. 1–6.

[69] K. L. McMillan, The SMV language, Tech. rep., Cadence Berkeley Labs (03 1999).

[70] P. Wolper, Temporal logic can be more expressive, in: Foundations of Computer Science, 1981, pp. 340–348.

[71] Texas Instruments, Keystone architecture - serial peripheral interface (spi) (03 2012).
URL https://web.archive.org/web/20180328180445/http://www.ti.com:80/lit/ug/sprugp2a/sprugp2a.pdf

[72] R. G. Michalko, Electrical starting, generation, conversion and distribution system architecture for a more electric vehicle (U.S. Patent US20060061213A1, 03 2006).

[73] I. Incer, M. Lohstroh, A. Iannopollo, E. A. Lee, A. Sangiovanni- Vincentelli, A Metric for Linear Temporal Logic, arXiv e-prints (2018) arXiv:1812.03923arXiv:1812.03923.

[74] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.

Appendix A. LTL satisfiability and validity as a model-checking problem

The model-checking problem can be formalized as a decision problem. Given a model M and property ϕ, the model checking problem answers whether M satisfies ϕ. If M does not satisfy ϕ, then the model checking algorithm generates a counterexample showing a trace from M that violates ϕ. Thus, model checking reduces to verify that the language generated by the model is a subset of the language of the property, $\mathcal{L}(M) \subseteq \mathcal{L}(\phi)$, or, equivalently, that $\mathcal{L}(M) \cap \mathcal{L}(\neg\phi) = \emptyset$ [74]. To verify that ϕ is a tautology, we then need to verify that $\mathcal{T} \cap \mathcal{L}(\neg\phi) = \emptyset$, where $\mathcal{T}$ is the set of all traces generated by an unconstrained model. To verify that ϕ is satisfiable, it is sufficient to check for the validity of $\neg\phi$. The counterexample generated in this last case will be a satisfiable assignment for ϕ.

In the rest of the section, we will describe the basic structure of an SMV program [69], and discuss how we can use a model checker supporting that language, i.e., nuXmv [38], to check for the satisfiability and validity of a certain LTL formula.

Appendix A.1. LTL Validity as an SMV *Program*

An SMV program is a collection of modules and a set of specifications over variables of those modules. Each program must have a module called `main`, which represents the starting point.

```
1   MODULE main
2   VAR bit0 : counter_cell(TRUE);
3       bit1 : counter_cell(bit0.carry_out);
4       bit2 : counter_cell(bit1.carry_out);
5   LTLSPEC G F bit2.carry_out
6
7   MODULE counter_cell(carry_in)
8   VAR value : boolean;
9   ASSIGN
10    init(value) := FALSE;
11    next(value) := value xor carry_in;
12  DEFINE  carry_out := value & carry_in;
```

Figure A.7: An SMV program implementing a 3-bit counter[6].

The listing in Figure A.7, which models a 3-bit counter, shows the typical structure of an SMV program. Each module defines a set of variables that can have pure types (Boolean, Integers, etc.), or be an instance of another module in the program. Modules can accept parameters, which are passed by reference, and all of their internal variables can be explicitly accessed by their parent module or a specification through the dotted notation *‹module instance›.‹variable›*, as it happens, for instance, in lines 3 or 5. Referencing variables without the dotted notation will result in accessing them according to their scope with respect to the caller.

[6]Program adapted from `http://nusmv.fbk.eu/NuSMV/userman/v11/html/nusmv_2.html`.

Each module can define what is the valid evolution of its variables and parameters over time by defining transition relations, which are Boolean relations involving current and next-state variables, or explicit variable state assignment. In the example, an explicit state assignment is shown in the section starting from line 9 of the example. There, the variable `value` is first initialized and then updated to implement the module's behavior. If a variable is not assigned to any module, or if its behavior is not specified by a transition relation, its behavior is nondeterministic.

To verify a certain LTL property, introduced by the identifier `LTLSPEC` (e.g., see line 5), the model checker runs all the modules synchronously and verifies that the constraint defined in the property holds. If it doesn't, it generates a trace that shows one system run in which all the variables behave according to the implementation but the property is not satisfied. We refer the reader to [38] for a full description of the SMV language.

Appendix A.1.1. Checking LTL validity and satisfiability

Given an LTL formula ϕ over variables in a set $\mathcal{V}$, one can check for its validity using NuXmv by creating an SMV program with only one module, `main`, instantiating all the variables in $\mathcal{V}$. The module, however, is not required to specify any behavior over those variables, as any value will be valid. Additionally, the program will have ϕ as its only `LTLSPEC` constraint.

When model-checking the program, NuXmv will either return *True*, meaning that ϕ is indeed valid, or it will return a counterexample showing why the formula is not valid. Instead, to check whether ϕ is satisfiable, we simply need to check for the validity of its negation, $\neg\phi$. If the model checker returns *True*, it means that the formula is not satisfiable. If the formula is satisfiable, then the counterexample generated by the model checker will represent an assignment for the variables in $\mathcal{V}$ that satisfies ϕ.

Under the hood, NuXmv computes the language associated with the main module, $\mathcal{L}(\texttt{main}) = \mathcal{T}$. Since the module does not specify any behavior, its language corresponds to the set of all behaviors $\mathcal{T}$. On the other side, the model checker computes the language of the negation of the specification, $\mathcal{L}(\neg\phi)$. If $\mathcal{L}(\texttt{main}) \cap \mathcal{L}(\neg\phi) = \emptyset$, then ϕ is valid. Otherwise, the result is used to derive a counterexample which is shown to the user.

Example Appendix A.1 (SMV Program for LTL Validity Check)**.** *Let $\phi = \square(a \wedge b) \rightarrow \diamond a$ be an LTL formula over variables in $\mathcal{V} = \{a, b\}$. To check for its validity with NuXmv, we will need to model-check the program in Figure A.8, which is obviously true.*

```
1    MODULE main
2    VAR a : boolean;
3        b : boolean;
4
5    LTLSPEC G (a & b) -> F a
```

Figure A.8: The SMV program for checking the validity of $\phi = \square(a \wedge b) \rightarrow \diamond a$.

```
1  -- specification  G bit2.carry_out  is false
2  -- as demonstrated by the following execution sequence
3  Trace Description: LTL Counterexample
4  Trace Type: Counterexample
5    -- Loop starts here
6    -> State: 1.1 <-
7      bit0.value = FALSE
8      bit1.value = FALSE
9      bit2.value = FALSE
10     bit0.carry_out = FALSE
11     bit1.carry_out = FALSE
12     bit2.carry_out = FALSE
13   -> State: 1.2 <-
14     bit0.value = TRUE
15     bit0.carry_out = TRUE
16   ⋮
17   -> State: 1.9 <-
18     bit0.value = FALSE
19     bit1.value = FALSE
20     bit2.value = FALSE
21     bit0.carry_out = FALSE
22     bit1.carry_out = FALSE
23     bit2.carry_out = FALSE
```

Figure A.9: Counterexample generated by model-checking the program in Figure A.7 with the specification `G bit2.carry_out`. The dots in Line 16 have been added manually to indicate the shortening of the trace.

Appendix A.1.2. Structure of a NUXMV *Counterexample*

The listing in Figure A.9 shows the structure of a counterexample generated by NUXMV. In this case, we model-checked the program in Figure A.7 changing the LTL specification to `G bit2.carry_out`. The trace shows a sequence of states (e.g., see line 6), where each state corresponds to a step of the whole system. Each state displays the evaluation of all the variables in the program in that particular step, using the dotted notation seen in Figure A.7.

Counterexamples can either be finite or infinite. If they are infinite, they will have a lasso-shaped structure, meaning that at, a certain point, the trace will form a loop. Thus, after a (possibly empty) initialization sequence, the system behavior will be cyclic. The beginning of a loop is clearly indicated in the trace in Figure A.9 (e.g., see line 5).