Specification Decomposition for Synthesis from Libraries of LTL Assume/Guarantee Contracts

Antonio Iannopollo*, Stavros Tripakis†, Alberto Sangiovanni-Vincentelli*

*EECS Department
University of California, Berkeley
Berkeley, USA

Email: {antonio, stavros, alberto}@berkeley.edu

†Department of Computer Science
Aalto University, Aalto, Finland

Abstract—Contract-Based Design is a methodology that allows for compositional design of complex systems. Given a contract representing a specification, it is possible to formally satisfy it by composing a number of simpler contracts. When these simpler contracts are chosen from a library of existing solutions, we talk about synthesis from contract libraries. There are techniques to automate the synthesis process, but they are computationally intensive, especially for complex specifications. In this paper, we describe an efficient technique to partition a specification, i.e., an LTL-based Assume/Guarantee contract, in a number of simpler sub-specifications which can be satisfied independently. Once all these smaller problems are solved, it is possible to safely merge their solutions to satisfy the original specification. We show the effectiveness of our technique in an industrial case study.

I. INTRODUCTION

Traditional design methodologies are struggling to keep up with the rising complexity of modern Cyber-Physical Systems (CPS). Contract-Based Design (CBD) [1]–[3] allows designers to manage it by enabling compositional design within a rigorous theoretical framework. Dealing with formal specifications, however, poses a challenge to human designers. Developing tools and techniques to mitigate the rigidity of formal languages is, indeed, a necessity.

Given a specification described by a contract, it is possible to satisfy it by composing a number of simpler contracts, where CBD provides the mathematical tools required to validate the design. When contracts are automatically chosen from a library of predefined designs, we talk about Synthesis from Component Libraries (SCL) [4]–[7]. Although synthesis tools and algorithms have come a long way, SCL remains a computationally hard problem.

In this paper, we present a way to increase the scalability of synthesis from libraries of components defined by Linear Temporal Logic-based (LTL) Assume/Guarantee (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. The result is an algorithm able to generate a set of sub-specifications which are as “small” as possible, only requiring a number of operations quadratic in the number of variables of the original contract.

The rest of the paper is organized as follows. In Section II we present related work. In Section III, we lay out the theoretical elements to support our work, and provide a formal analysis of our approach in Section IV. Then, in Section V, we describe in detail our decomposition algorithm. In Section VI, we evaluate the performance of our method by applying it to an industrial case study, i.e., the design of the control software for an Aircraft Electrical Power System (EPS), and draw conclusions in Section VII.

II. RELATED WORK

In [4], the authors define a general framework for synthesis from component libraries, where components are appropriately constrained to achieve decidability. The solution proposed in that paper uses a constraint solver to build a candidate composition as a network of components, and a verifier that is able to evaluate the semantics of the candidate. This paper focuses on maximizing the effectiveness of such synthesis techniques—specifically, when components are described by LTL A/G contracts.

The problem of specification decomposition is not new. In [8], Henzinger et al. 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. We, on the other hand, are interested in synthesis, therefore the structure of the design is not known a priori. We share, however, the idea of using the specification itself as a placeholder for other components.

In [9], given a set of components and a safety specification, the goal is to generate a set of minimally restrictive assumptions (one per component). Such assumptions, found through a fixed point computation, are used to synthesize controllers for the components. Our goal is similar: breaking down the synthesis process in simpler sub-tasks. In our case, however, the decomposition in sub-problems depends solely on the specification.
III. PRELIMINARIES

A. LTL A/G Contracts

A/G contracts represent a specific instance of the more general contract theory [1], where the behavior of a component, i.e., its promise or guarantee, and its expectations from the environment, i.e., its assumption, are expressed using assertions. Here, A/G contracts are defined using synchronous assertions, which are sets of behaviors. A behavior is a sequence of evaluations of variables from a fixed alphabet \( \Sigma_{IO} \) sharing the same domain.

Formally, an A/G contract is a pair \( C = (A,G) \) where \( A \) and \( G \) are synchronous assertions representing assumptions and guarantees, respectively. We concretely express the sets \( A \) and \( G \) as a pair of LTL formulas, \( \varphi_A \) and \( \varphi_G \), each denoting the set of all traces (behaviors) that satisfy it. As for \( A \) and \( G \), also \( \varphi_A \) and \( \varphi_G \) are defined over the same set of variables \( \Sigma_{IO} \).

Contract theory includes a number of operations to manipulate contracts, including (parallel) composition and refinement. The composition of two contracts \( C_1 = (\varphi_{A1}, \varphi_{G1}) \) and \( C_2 = (\varphi_{A2}, \varphi_{G2}) \) can be directly defined in terms of LTL formulas as

\[
C_1 \circ C_2 = ((\varphi_{A1} \land \varphi_{A2}) \land \neg(\varphi_{G1} \land \varphi_{G2}) \land \varphi_{G1} \land \varphi_{G2}).
\]

Contract composition is associative and commutative.

Refinement, instead, formalizes a notion of substitutability and it is defined as a preorder on contracts. A contract \( C_1 = (\varphi_{A1}, \varphi_{G1}) \) refines contract \( C_2 = (\varphi_{A2}, \varphi_{G2}) \), written \( C_1 \preceq C_2 \), iff

\[
\varphi_{A2} \Rightarrow \varphi_{A1} \quad \text{and} \quad \varphi_{G1} \Rightarrow \varphi_{G2}
\]

are both valid. Refinement can be efficiently verified, for LTL A/G contracts, using any tool able to check satisfiability of LTL formulas, such as a model-checker [10].

B. Synthesis from Component Libraries

In its simplest form, a component \( K \in \mathbb{K} \), where \( \mathbb{K} \) is the domain representing the space of all possible components, is a tuple \( K = (I_K, O_K, C_K) \). \( I_K \) is the set of input ports, \( O_K \) is the set of output ports, and \( C_K \) is an A/G contract that describes the component behavior. Variables in \( C_K \) correspond to ports in \( I_K \) and \( O_K \). \( I_K, O_K \) and \( C_K \) are all defined over a common set of symbols, or alphabet, \( \Sigma_{IO} \).

A library is a pair \( L = (Z, R) \). Here \( Z = \{K_1, ..., K_n\} \) is a finite collection of components. In \( Z \), several components can have the same specification \( C \), but they are required to have at least unique names for ports (and thus variables). \( R \) is a set of constraints defining how components can be connected to each other. Constraints in \( R \) are defined over ports of all components in \( Z \).

We formalize connections by applying a renaming function to ports. For a component \( K \) we define its renaming function as

\[
\rho^K : 2^{\Sigma_{IO}} \times \Sigma_{IO} \rightarrow \mathbb{K}
\]

that returns a new component. For instance, given a component \( K_1 \) and a set of port pairs \( M = \{p,q\} \), we use

\[
\rho^{K_1}(M) = K_1' \quad \text{to indicate a component where the port "p" has been renamed "q". If another component, say } K_2, \text{ has a port named q, then we say that } K_1' \text{ and } K_2 \text{ are connected through q, or simply that } K_1.p \text{ and } K_2.q \text{ are connected. If a port } t \text{ doesn't share its name with any other port, we say that } t \text{ is unconnected.}
\]

The composition of two components \( K_1 = (I_1, O_1, C_1) \) and \( K_2 = (I_2, O_2, C_2) \), defined iff \( O_1 \cup O_2 = \emptyset \), is a new component \( K_1 \circ K_2 = ((I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, C_1 \circ C_2) \), meaning that input ports that are connected to output ports are considered outputs in the resulting composition.

We say that a component \( \rho^{K_1}(M_1) = K_1' = (I_1', O_1', C_1') \) refines a component \( \rho^{K_2}(M_2) = K_2' = (I_2', O_2', C_2') \), written \( K_1' \preceq K_2' \), for some renamings \( M_1, M_2 \), if and only if

\[
I_1' \subseteq I_2', O_1' \subseteq O_1', \text{ and } C_1' \preceq C_2'.
\]

We consider the system specification \( S = (I_S, O_S, C_S) \), that needs to be synthesized, as a component itself.

Example 1 (Component Connection). Let

\[
K_1 = (I_1, O_1, C_1) = ((\{a\}, \{b\}, ((a) \land (\diamond a), (\diamond b)))
\]

\[
K_2 = (I_2, O_2, C_2) = ((\{c\}, \{d\}, ((c) \land (\diamond c), (\diamond c \Rightarrow d)))
\]

be two components and \( M = \{(b, c)\} \) a set of port pairs. Then renaming \( K_1 \) according to \( M \) will yield a component

\[
\rho^{K_1}(M) = K_1' = ((\{a\}, \{c\}, (\Box (a), (\diamond c)))
\]

Therefore, we say that \( K_1' \) and \( K_2' \) are connected through \( c \).

The concepts we just introduced are useful to describe the problem of synthesis from component libraries:

Definition 1 (Problem of Synthesis from Component Libraries (SCL)). Given a specification expressed as a component \( S \), and a library \( L = (Z, R) \), the goal is to find a set of components \( K = \{K_1, ..., K_n\} \subseteq Z \), and a set of renamings \( M = \{M_1, ..., M_n\} \) such that

1. All the constraints in \( R \) are satisfied;
2. The composition of components in \( K \), renamed according to \( M \), refines the specification \( S \):

\[
\rho^{K_1}(M_1) \circ \cdots \circ \rho^{K_n}(M_n) \preceq S
\]

In [4], a solution to this problem has been implemented for libraries of components described by LTL A/G contracts, using a Satisfiability Modulo Theories (SMT) solver to satisfy the constraints in \( R \) (point 1. above), and a model-checker to verify the refinement (point 2.).

In this paper, we present a technique to improve the scalability of the synthesis process, which is reduced to a series of simpler tasks. In Section IV-A, we will introduce a slightly modified synthesis problem that can still be solved, however, using the approach presented in [4]. The technique we present here focuses on contracts. For simplicity, in the next sections we will refer directly to contracts instead of components, although we will always keep in mind the framework described in this section. Through a slight abuse of notation, we will apply the renaming functions directly to contracts, and we...
will refer to inputs and output variables of a contract $C$ using the sets $I_C$ and $O_C$, as we normally would do for components.

IV. CONTRACT DECOMPOSITION

Given a system specification expressed as a contract, our objective is to decompose it in several sub-specifications (or projections), to simplify the synthesis problem in Definition 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.

To introduce our goal, we use the notion of projection for a LTL $A/G$ contract, which will be defined later:

**Definition 2 (Contract Decomposition Problem (CD)).** Let $\Pi_V(C)$ indicate the projection of a contract $C$ over the set of output variables $V \subseteq O_C$. The CD problem consists in partitioning $O_C$ into $n$ sets of variables $V_1, ..., V_n$, with $n \geq 1$, such that

$$\Pi_{V_1}(C) \otimes \Pi_{V_2}(C) \otimes \ldots \otimes \Pi_{V_n}(C) \leq C$$

(6)

In Definition 2, we are searching for a partition of the output variables of $C$. As we will see in the next paragraphs, composing projections results in a contract with more outputs than the initial one. Looking at the definition of refinement for components in Equation 4, we see that adding output variables in the components on the left-hand side of the refinement operation is not a problem. Conversely, we cannot claim the same for input variables. Focusing only on output variables guarantees that the refinement between components is always satisfied if the refinement between contracts in Equation 6 holds.

As mentioned above, Definition 2 requires a well-defined projection operation. Unfortunately, in [11] Wolper proves that LTL formulas are not, in general, closed under projection. Therefore, we cannot define the projection operation for LTL $A/G$ contracts by simply taking the projection of their assumption and guarantee formulas.

The following definition of projection for LTL $A/G$ contracts, however, doesn’t involve projecting LTL formulas and still allows for the analysis of the CD Problem:

**Definition 3 (Projection of LTL $A/G$ Contracts).** Given a LTL $A/G$ contract $C$ and a subset of its output variables $V \subseteq O_C$, its projection with respect to $V$ is a contract

$$\Pi_V(C) = \rho^C(M_V)$$

(7)

where $M_V$ specifies the following renamings:

$$\forall p \in O_C : p \not\in V \Leftrightarrow (p, x) \in M_V$$

(8)

with $x \in \Sigma I_O \setminus (I_C \cup O_C)$ being a fresh variable.

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

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

$$\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \leq C$$

(9)

where $\bar{V} = \{ p | p \in O_C \setminus V \}$ 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 2** (Not all projections are valid). Consider the contract

$$C = (\varphi_A, \varphi_G) = (\text{true} \square (a \lor b))$$

and the set $X = \{a\}$. The projection associated with $X$ is $\Pi_X(C) = (\text{true}, \square (a \lor b))$ while its complement is $\Pi_{\bar{X}}(C) = (\text{true}, \square (a_0 \lor b))$. To verify the validity of the projection, we need to verify that $\Pi_X(C) \otimes \Pi_{\bar{X}}(C) \leq C$ holds.

Let us start by verifying that the guarantees of the composition are more constrained than $\varphi_G$:

$$\square (a \lor b) \land \square (a_0 \lor b) \Rightarrow \square (a \lor b)$$

The equation above is not always true. Consider, for instance, the case in which, at time 0, $a = \text{false}$, $b_0 = \text{true}$, $a_0 = \text{true}$, and $b = \text{false}$. Thus, the projection $\Pi_X(C)$ is not valid. For another contract $C’ = (\text{true}, \square (a \lor b))$, instead, the set $X = \{a\}$ yields a valid projection.

Before proceeding, it is useful introducing the concept of independent variables for a certain LTL formula:

**Definition 4 (Independent Variables).** Let $\varphi$ be a LTL formula over a set of variables $P$. Let also $V \subseteq P$. We say that variables in $V$ are independent in $P$ for $\varphi$ if and only if, for each sequence $\sigma$ of evaluations of variables in $P$ that falsifies $\varphi$, then $\varphi$ can also be falsified only by the sequence $\sigma_V$ of evaluations of variables in $V$ or the sequence $\sigma_{\bar{V}}$ of evaluations of variables in $V = P \setminus V$:  

$$\forall \sigma : \sigma |_V \Rightarrow \sigma_V \land \sigma_{\bar{V}} \not\Rightarrow \varphi$$

(10)

**Example 3.** Let $\varphi = \square (a \lor b)$. Then $V = \{a\}$ does not contain independent variables$^2$. Consider, for instance, the finite sequence $\sigma = [(a = \text{false}, b = \text{false})]$. $\varphi$ falsifies $\varphi$, but $\sigma_V = [(a = \text{false})]$ does not, because the evaluation of $b$ is unknown. On the other hand, for $\varphi’ = \square (a \land b)$ 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 described in Section V.

**Theorem IV.1.** Let $C = (\varphi_A, \varphi_G)$ be a LTL $A/G$ contract, and consider a subset of its output variables $V \subseteq O_C$. If variables in $V$ are independent in $O_C$ for $\varphi_G$, then the projection $\Pi_V(C)$ is valid.

**Proof.** Consider a contract $C = (\varphi_A, \varphi_G)$ and a subset of variables $V \subseteq O_C$, where $V$ contains independent variables. To verify the validity of the projection $\Pi_V(C)$, $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \leq C$ must hold. This means verifying that:

1) $\varphi_A \Rightarrow \varphi_{AV} \land \varphi_{AV} \neg (\varphi_{GV} \land \varphi_{GV})$

2) $\varphi_{GV} \land \varphi_{GV} \Rightarrow \varphi_{G}$

We start from point 2). Let $\sigma_G$ be a sequence of evaluations of output variables in $C$ that falsifies $\varphi_G$. Then, by definition of

$^2$If the context is clear, as in this case, we will just say that variables in $V$ are independent.
1) in Theorem IV.1. Hence, the theorem is proved.

For point 1), let $\sigma_A$ be a sequence of evaluations of input variables in $C$ that falsifies the formula on the right-hand side of the implication, $\varphi_{AV} \land \varphi_{AV} \land \neg(\varphi_{GV} \land \varphi_{GV})$. This means that $\sigma_A$ falsifies also the stronger formula $\varphi_{AV} \land \varphi_{AV}$. Recall that $C$, $\Pi_V(C)$, and $\Pi_G(V)$ all share the same input variables. Thus, any $\sigma_A$ that falsifies $\varphi_{AV} \land \varphi_{AV}$ will also falsify $\varphi_A$. Hence, 1) is always true, too. This proves the theorem.

A. Using Projections for Synthesis

At this point, we know how to find valid projections for a contract. In this section, we describe how such projections can be used to simplify the problem of synthesis from component 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 6. The following theorem clarifies this point.

**Theorem IV.2.** Let $\Pi_V(C),...,\Pi_V(C)$ be valid projections of a contract $C$. If $V_1,...,V_n$ all contain variables independent in $O_C$ for $\varphi_G$, and $V_1 \cup \ldots \cup V_n = O_C$, then

$$\Pi_V(C) \otimes \ldots \otimes \Pi_V(C) \preceq C$$

(11)

**Proof.** By the definition of valid projection in Equation 9, we know that for each projection $\Pi_V(C)$ of a contract $C = (\varphi_A, \varphi_G)$, the refinement $\Pi_V(C) \otimes \Pi_V(C) \preceq C$ holds, where $V_i \cup \overline{V_i} = O_C$ by construction. To prove Equation 11 we need to verify that:

1) $\varphi_A \Rightarrow \varphi_{AV_1 \land \ldots \land \varphi_{AV_n} \land \neg(\varphi_{GV_1 \land \ldots \land \varphi_{GV_n})$.

2) $\varphi_{GV_1 \land \ldots \land \varphi_{GV_n} \Rightarrow \varphi_G$.

The proof, at this point, is similar to the proof of Theorem IV.1. We start from point 2). For $\sigma_G$ being a sequence of evaluations of variables in $O_C$ which falsifies $\varphi_G$, we know that there must exist a $V_i \subseteq O_C$ with independent variables such that the guarantees $\varphi_{V_i}$ of the projection $\Pi_V(C)$ are false, too. Without loss of generality, let us assume that $V_i$ is also minimal, i.e., there not exists a proper subset of $V_i$ which also contains independent variables. Since $\{V_1,...,V_n\}$ all contain independent variables and $V_1 \cup \ldots \cup V_n = O_C$, then there exists a $V_i \subseteq \{V_1,...,V_n\}$ such that $V_i \preceq V_i$. Thus $\varphi_{V_i}$ will be falsified by $\sigma_G$, 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 IV.1. Hence, the theorem is proved.

Theorem IV.2 explains how we can partition a contract $C$ using $n$ valid projections, $\Pi_V(C),...,\Pi_V(C)$. This is a good news, because now we can synthesize a composition of contracts that refines $C$ from a library $L$, if such composition exists, by independently synthesizing compositions for the projections $\Pi_V(C),...,\Pi_V(C)$. That is, if there exist compositions such that

$$C_{i}^{V_1 \otimes \ldots \otimes C_{m_i}^{V_i}} \preceq \Pi_V(C)$$

for some $m_1,...,m_n$, then by the independent development property in [1], the following holds:

$$\bigotimes_{1 \leq i \leq n} (C_{i}^{V_1 \otimes \ldots \otimes C_{m_i}^{V_i}} \preceq \Pi_V(C) \preceq C$$

(12)

However, we are not done yet. Each projection $\Pi_V(C)$, in fact, has exactly the same number of variables than $C$, although now we are only interested in the subset $V_i$. Thus, we need a way to limit synthesis only to $V_i$. We solve this issue by defining a variant of the SCL problem in Definition 1. Note that, for consistency, in the definition below we use again the concept of component introduced earlier.

**Definition 5** (Problem of Partial Synthesis from Component Libraries (PSCL)). Let the component $S = (I_S, O_S, C_S)$ be a system specification, $V$ be a subset of independent variables $V \subseteq O_S$, and define $\overline{V} = O_S \setminus V$. Define also the projection $C_S = \Pi_V(C_S)$ and its associated component $S = (I_S, O_S, C_S)$. Given a library $L = (\mathcal{I}, \mathcal{R})$, find a set of components $K = \{K_1,...,K_m\} \subseteq \mathcal{I}$, and a set of renamings $M = \{M_{1},...,M_{m}\}$ such that

1) All the constraints in $\mathcal{R}$ are satisfied;
2) The following holds:

$$\rho^{K_1}(M_{1}) \otimes \ldots \otimes \rho^{K_m}(M_{m}) \otimes \overline{S} \preceq S$$

(13)

In a nutshell, to find a solution for a contract $C_S$ and a set $V \subseteq O_C$, we ask the synthesizer to find a set of components $\rho^{K_1}(M_{1}) \otimes \ldots \otimes \rho^{K_m}(M_{m})$, such that their contract composition satisfies $C_S \preceq \Pi_V(C_S) \preceq C_S$. The synthesizer is forced to use $\overline{S}$ as a placeholder connected to ports in $V$, therefore it avoids wasting time in satisfying constraints for ports that are not in $V$. Overall, this results in n smaller synthesis problems which can be run independently, i.e., concurrently, using techniques such as the one proposed in [4].

Once we have found a solution for all the projections, Equation 12 guarantees that putting all the pieces back together will result in a proper refinement of our original system specification.

V. AN EFFICIENT DECOMPOSITION ALGORITHM

In this section we discuss an efficient algorithm to decompose a contract $C$ following the concepts discussed in Section IV. One obvious possibility would be to exhaustively check whether $\Pi_V(C) \otimes \Pi_V(C) \preceq C$ holds for all the possible $V_i \subseteq \overline{O_C}$, where $\overline{O_C}$ is the powerset of $O_C$. This is not very efficient, as it requires checking Equation 9 at least $2^{|O_C|}$ times. We propose a better solution which 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 $V_i$ till it contains independent variables. We do so by analyzing the counterexamples obtained verifying some ad hoc formulas.

Figure 1 describes the main decomposition algorithm. For each output variable $p$ (line 2), we start with a set $V$ containing only $p$ (3). After creating the candidate projections
Algorithm 1: DecomposeContract

Input: Contract \( C = (\phi, \varphi_C) \)
Output: Set of valid contract projections

1. \( \text{clusters} \leftarrow \{\} \); 
2. for \( p \in O_C \) do 
3. \( V \leftarrow \{p\} \); 
4. passed \( \leftarrow \) FALSE; 
5. repeat 
6. \( V \leftarrow O_C \setminus V \); 
7. rightF \( \leftarrow \text{RefFormula}(\Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C) \); 
8. leftF \( \leftarrow \text{IsolateVars}(V) \); 
9. passed, trace \( \leftarrow \text{Verify}(\text{leftF} \Rightarrow \text{rightF}) \); 
10. if not passed then 
11. \( \text{dvar} \leftarrow \text{ParseTrace} \big( \text{trace} \big) \); 
12. \( V \leftarrow V \cup \{\text{dvar}\} \); 
13. end 
14. until passed; 
15. clusters \( \leftarrow \text{clusters} \cup V \); 
16. end 
17. \( \text{clusters} \leftarrow \text{MergeClusters} \left( \text{clusters} \right) \); 
18. return \( \{\Pi_V(\phi) | \forall V \in \text{clusters} \} \)

Fig. 1. Contract Decomposition algorithm. It takes a contract \( C \) as input, and returns a set of \( n \) projections such that \( \Pi_V(\phi) \otimes \cdots \otimes \Pi_V(\varphi_C) \leq C \).

\( \Pi_V(\phi) \) and \( \Pi_V(\varphi_C) \), the algorithm generates the formula to verify. This formula consists of an implication (line 9), where the right hand side formula is the expression refining \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) (line 7), as described in Equations 1 and 2. The left-hand side formula is generated by the \( \text{IsolateVars} \) function (line 8). It computes the disjunction:

\[
\bigvee_{q \in V} \bigwedge_{r \in V \setminus \{q\}} \Box(\Pi_V(\phi).t = \Pi_V(\varphi_C).t) \tag{14}
\]

The intent of Equation 14 is to allow the model-checker to prove that the refinement \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) does not hold, i.e., the set \( V \) is too small, by opportunistically simulating connections between \( \Pi_V(\phi) \) and \( \Pi_V(\varphi_C) \). For each variable \( q \in V \), we build a conjunction stating that all the variables in \( V \) but \( q \) need to behave in the same way for both \( \Pi_V(\phi) \) and \( \Pi_V(\varphi_C) \), hence simulating a connection between them. The leftmost disjunction, then, tells the model-checker that any of those conjunctions needs to hold.

Going back to the algorithm in Figure 1, in line 9 the formula \( \text{leftF} \Rightarrow \text{rightF} \) is verified using a model-checker. Now, if \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) holds, then \( \text{leftF} \Rightarrow \text{rightF} \) will be true, meaning that \( \Pi_V(\phi) \) is a valid projection, and we can move on to the next iteration (line 15). If, however, \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) can be falsified, the model-checker will generate a counterexample that proves that \( \text{leftF} \) is true while \( \text{rightF} \) is false. Considering the construction of \( \text{leftF} \), this means that only one variable in \( V \) has been used to falsify the formula, as all the others are forced to behave as if they were connected to the same contract. We can then analyze the counterexample to identify such variable and add it to \( V \) (lines 11 and 12), and repeat the process till \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) is valid. Finally the last step of DecomposeContract, line 17, guarantees that the set \( V_1, \ldots, V_n \) is a partition of \( O_C \), as required by Definition 2. The algorithm always terminates, as in the worst case we have that \( V = O_C \), thus \( \Pi_V(\phi) = C \) and \( V = \emptyset \), which always verifies \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \). It does so invoking the model-checker (through the function \( \text{Verify} \)), which has exponential complexity) at most \( n^2 \) times, where \( n \) is the number of output ports of \( C \).

A. Algorithm DecomposeContract in Figure 1 is sound and complete

To show that the algorithm is sound, 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 \( \text{Verify} \) in DecomposeContract, line 9, returns true if \( \text{rightF} \) is true. If so, then \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) holds and the projection over \( V \) is valid. We say so because otherwise the model-checker would have found a trace in which \( C \) is false, but \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \) is not, according to Definition 4. If \( \text{rightF} \) is false, then it means that the variables in \( V \) are not independent, i.e., \( V \) is too small. Therefore, the variables in \( V \) depend on (at least) a variable in \( \bar{V} \). Equation 14 helps us identify such variable by allowing the model-checker to simulate connections between \( \Pi_V(\phi) \) and \( \Pi_V(\varphi_C) \). Equation 14, in fact, allows only one variable in \( V \) to behave as if it were not controlled by \( \Pi_V(\phi) \). The model-checker, thus, will provide a counterexample in which exactly one variable will cause the formula \( \text{leftF} \Rightarrow \text{rightF} \) to fail. This variable must be dependent on variables on \( V \), because otherwise \( \text{rightF} \) would be true. By parsing the counterexample trace we can recognize such variable and add it to \( V \). This process stops when \( \Pi_V(\phi) \otimes \Pi_V(\varphi_C) \leq C \) is true, which means, as we have seen above, that \( V \) contains independent variables. In the worst case, the process stops when \( V = O_C \), which will always result in a valid projection. The algorithm is therefore sound.

The algorithm is also complete, meaning that if there exists a partition \( V_1, \ldots, V_n \) such that \( \Pi_{V_1}(\phi) \otimes \cdots \otimes \Pi_{V_n}(\varphi_C) \leq C \), with \( n > 1 \), then the algorithm will find it. The procedure of letting the model-checker pick a single variable to add to \( V \) at each iteration, indeed, guarantees that if there exists a proper subset \( V \subset O_C \) with independent variables we will find it.

VI. THE AIRCRAFT ELECTRICAL POWER SYSTEM EXAMPLE

We implemented the proposed algorithm in a modified version of pyco, the tool described in [4]. All the experiments were run on a 2.5 GHz Intel Core i7 machine, with 16GB of RAM.

Figure 2 shows a simplified structure of an aircraft EPS [12], [13]. Generators deliver power to the loads via AC and DC buses. In case of generator failures, Auxiliary Power Units (APUs) provide the required power. The power flow from sources to loads is determined by contactors, i.e., electromechanical switches, that can be opened or closed. Transformer Rectifier Units (TRUs) convert and route AC power to DC buses.

The function of the controller, called Bus Power Control Unit (BPCU), is to react to failures (of Generators, APUs, and TRUs) and reroute power by actuating the contactors. In our model, all variables are Boolean, corresponding to component faults and contactor states (open or closed). Our goal is to
synthesize the logic of the BCPU from a set of subsystem controllers, described by a library of 20 LTL $A/G$ contracts.

Table I represents the system specification. Input ports reflect the status of EPS elements (such as generators), while output ports represent contactors. The assumptions capture the expectation that when a component fails, it will not be operational again. The guarantees include the promise that faulty generators will be isolated, no short-circuit will happen, and loads will always be powered.

We ran 9 synthesis tasks with increasing complexity (each task was an incremental subset of the Guarantees in Table I), both with and without the application of the contract decomposition procedure. Figure 3 illustrates the results we obtained. A typical solution satisfying all the specifications consisted of 6 components. As expected, decomposing the specification leads to better performance, resulting in roughly one order of magnitude faster synthesis (especially when the complexity of the specification increases).

When decomposing the full specification, the decomposition algorithm generated 7 sets of independent variables: $V_1 = \{C_1\}, V_2 = \{C_4\}, V_3 = \{C_2, C_5, C_6, C_9\}, V_4 = \{C_7\}, V_5 = \{C_8\}, V_6 = \{C_9\}$ and $V_7 = \{C_{10}\}$.

VII. Conclusion

In this paper, we presented a technique to increase scalability of synthesis from component libraries for components that are described by LTL $A/G$ contracts. We defined the notions of contract decomposition and projection, and described an efficient algorithm to perform such decomposition. We are currently planning to extend this work by reducing the complexity of the verification step in the DecomposeContract algorithm, and by exploring ways to further increase the parallelization of the synthesis process.

REFERENCES


