A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications

Antonio Iannopollo

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-173
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-173.html

December 14, 2018

Acknowledgement

The work in this dissertation and related publications has been supported by the National Science Foundation (NSF), via the project "ExCAPE: Expeditions in Computer Augmented Program Engineering" (CCF-1139138), 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.

A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications

by

Antonio Iannopollo

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Alberto L. Sangiovanni-Vincentelli, Chair
Professor Sanjit A. Seshia
Professor Francesco Borrelli

Fall 2018

A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications

Abstract

A Platform-Based Approach to Verification and Synthesis of Linear Temporal Logic Specifications

by

Antonio Iannopollo

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto L. Sangiovanni-Vincentelli, Chair

The design of Cyber-Physical Systems (CPS) is challenging as it requires coordination across several domains (e.g., functional, temporal, mechanical). To cope with complexity, rarely a CPS is built from scratch. Instead, it is assembled by reusing available components and subsystems. If a component is not available, then it is made to order according to a specification which ensures its compatibility with the rest of the system.

To achieve design goals faster while guaranteeing system safety, the correct instantiation of modules and subsystems is essential. Formal specifications, such as those expressed in Linear Temporal Logic (LTL), have the potential to drastically reduce design and implementation efforts by enabling rigorous requirement analysis and ensuring the correct composition of reusable designs. Composing formal specifications, however, is a tedious and error-prone activity, and the scalability of existing formal analysis techniques is still an issue.

In this dissertation, we present a set of techniques and algorithms that leverage compositional design principles to enable faster verification and correct-by-construction, platform-based synthesis of LTL specifications. In our framework, a design is a composition of several components (which could describe both hardware and software elements) represented through their specifications, expressed as LTL assume/guarantee interfaces, or contracts. The collection of all the available contracts, i.e., a library, describes the design platform. The contracts in the library are the building blocks of different possible designs, and they are simple enough that their correctness can be easily verified, yet complete enough to guarantee the correct and safe use of the components they represent.

Our contribution is two-fold. On the one hand, we address the verification task: given an existing composition of contracts from the library, we want to check whether it satisfies a set of desired requirements. We improve the scalability of existing verification techniques by leveraging pre-verified relations between contracts in the library. On the other hand, we enable specification synthesis: given a (possibly incomplete) set of desired system properties, we are able to automatically generate a composition of contracts, chosen from a library, that satisfies them. We do so by devising a set of algorithms based on formal inductive synthesis, where a candidate is either accepted as correct or is used to infer new constraints and guide the synthesis process towards a solution. Additionally, we show how to increase the scalability of our approach by leveraging principles from the contract framework to decompose a synthesis problem into several independent tasks, which are simpler than the original problem. We validate our work by applying it to several industrial-relevant case studies, including the problem of verification and synthesis of a controller for the electrical power system of an aircraft.

To Alessandro Ingegnoli,
mentor and friend.

Contents

List of Figures

List of Tables

Acknowledgments

Being a Ph.D. student is like being in the middle of a journey. It can be difficult at times, but every step forward brings you new adventures, excitement, and awe. As with every journey, the people you walk with can make an immense difference in the way you remember the experience, even when you share with them only a small part of the path. I have been extremely lucky in meeting so many wonderful individuals along the way and, in these few lines, I want to thank them for their support and friendship.

First, I would like to thank my adviser, Prof. Alberto Sangiovanni-Vincentelli, who has been an extraordinary guide and inspiration in these past few years. He has always been there for me and, without his help, I would not be here today. He never ceases to amaze me with the breadth and depth of his knowledge and inquisitive mind.

Many thanks to Prof. Stavros Tripakis, who has been my unofficial co-adviser in my first years in Berkeley. I really enjoyed working with him, and I learned a lot during our discussions about computer science topics such as formal languages and interfaces, but also about politics and movies.

I am extremely grateful to Prof. Sanjit Seshia and Prof. Edward Lee, who have been a constant positive presence during these years. Sanjit, one of my first teachers here in Berkeley, introduced me to the beauty of formal methods. His brilliant work has greatly influenced my research. Edward has been a wonderful mentor. Among his many qualities, I admire his scientific acumen and the passion he puts in his work. His course on models of computation has been one of the most enjoyable I have ever taken. I would also like to thank Prof. Francesco Borrelli, who has served as a member of my qualifying exam and dissertation committees. His comments on my research were precious.

I have had the fortune to meet, here in Berkeley, some of the most brilliant students, researchers, and engineers in the world. I am thankful to call many of them great friends of mine. Thanks to Nikunj Bajaj, Alberto Puggelli, Marten Lohstroh, Shromona Ghosh, Tommaso Dreossi, Ben Zhang, Alberto Tempia Bonda, Inigo Incer, Fabio Cremona, Ankush Desai, John Finn, Baruch Sterin, Baihong Jin, Eddie Kim, Ankush Desai, Xiangyu Yue, Hokeun Kim, and Daniela De Venuto. We spent a lot of time together, both inside and outside of UC Berkeley, and I enjoyed every minute. My gratitude also goes to Chung-Wei Lin, Liangpeng Guo, Mehdi Maasoumy, Chris Shaver, Christos Stergiou, Matt Weber, Gil Lederman, Ilge Akkaya, Rohit Ramesh, Tianshi Wang, Richard Lin, Tomi Raty, Garvit Juniwal, Anil Aswani, Maryam Bagheri, Elisabetta Alfonsetti, Yen-Sheng Ho, Pangun Park, Raaz Dwivedi, Wei Yang Tan, Rohit Sinha, Ben Caulfield, Daniel Fremont, Yi-Chin Wu, Nishant Totla, and all the other DOP center residents. You guys rock! I am also extremely grateful to Jessica Gamble and Shirley Salanio for helping me navigate the bureaucracy of the EECS department and being super supportive.

I would like to thank United Technologies Corporation (UTC) and the Camozzi Group for the collaboration opportunities, and for the chance to learn a lot and meet so many brilliant collaborators. Thanks especially to Luiz Bacellar, Clas Jacobson, Michael Regelski, Dario Ferrarini, Andrea Camisani, and all the Camozzi research team. I also would like to thank the National Science Foundation (NSF), the ICyPhy Consortium, the Terraswarm Research Center, and the John R. Mott Scholarship Foundation for their support.

My gratitude goes also to the many great teachers and mentors that I had in my life. I will always be immensely grateful to Alessandro Ingegnoli, to whom I dedicate this dissertation, for believing in me and being the one who started the sequence of events that brought me here today. Among many others, I would also like to give a special thank to Marco Sgroi, Prof. Daniel Bovet, and Lucio Ficara.

Finally, I want to thank my family for being so incredibly loving and supporting. To my wife, Lauren, goes the deepest gratitude for her love, enthusiasm, and patience during these years of unpredictable work hours and last minute schedules. I could not have asked for a better partner. You are my oasis in the desert, and make every day so much better! Thanks to Santiago, Enza, Christopher, Giovanni, Karin, Alan, Charlotte, Matthew, zia Maria, and zio Guglielmo. You are the best family one could ask for. And, above everything else, thanks to my parents, Caterina and Girolamo, for being so wonderful and for giving me the best chances in life. I wish to be to my children what you have been for me.

Chapter 1

Introduction

System design is a process that starts with a high-level idea of what the system should be and ends with 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, following a fractal-like process [72, 83]. At each step, more and more details about the final design are added, until there is enough information to build the system.

To reduce costs, and shorten time-to-market, modern design methodologies emphasize design reuse by leveraging platforms, which are libraries of parametric components which 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 are able to 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 [74, 72, 77].

When system complexity increases, especially for safety-critical systems, maintaining consistency between design iterations and level of abstractions 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 reviews sessions are costly and ineffective [23]. Formal languages, such as temporal logic [68, 52], 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 abstractions, 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.

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. Yet, especially when described by temporal constraints, scalability remains an issue even for simple designs. In this dissertation, we describe methods to handle complexity when design elements are described by temporal constraints, and improve scalability.

1.1 Dissertation Overview

In this dissertation, 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 [68, 80]. 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 characterized well, functionally, by LTL. Examples include software, communication protocols, motion planning, reactive systems, and digital components in general [12, 25, 7, 52, 65, 63, 86].

To successfully apply techniques and methodologies that support automated reasoning, these design elements need to interact with each other in a well-defined manner. Indeed, components first need a mechanism to be interconnected together 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 [75, 11, 10, 2, 62, 30] 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 Chapter 3, formalize notions such as compatibility (are there legal environment for a component?) and consistency (are there legal 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 instantiation of their elements. Our goal is to verify, for such contracts, the mapping process of PBD first, and then to automate it. 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 dissertation, we talk about verification and synthesis of LTL specifications.

At the core of the techniques and algorithms that we develop there is the capability of manipulating LTL A/G contracts and verifying refinement among them, which can be reduced to a model checking problem [33, 19]. Our first goal is to improve the performance of current approaches to the refinement checking problem when verifying the compliance to a specification of a composition of contracts, i.e., the system, provided by the user. In Chapter 5, we show how pre-verified refinement relations stored in the contract library allow for faster verification of specification compliance by the system, by building decremental abstractions. In this case, the library contains both basic components and several subsystems realized with those components. Building abstractions, then, reduces to a mapping problem. We show that this approach can improve dramatically the verification performance by applying it to a case study of industrial relevance, i.e., the realization of a controller for the Electrical Power System (EPS) of an aircraft. We describe the EPS problem in Chapter 4, and we will use it as a common example throughout the dissertation.

We, then, direct our attention to the synthesis of correct composition of contracts, proceeding stepwise. First, in Chapter 6, we concentrate on the problem of synthesis assuming that the only output of the refinement verification procedure is a simple yes/no answer. We call this problem “Constrained Synthesis from Component Libraries” (CSCL), and we devise a synthesis procedure based on the Oracle-Guided Inductive Synthesis (OGIS) paradigm [49, 48], where erroneous candidate solutions are used to infer new constraints to guide the synthesis process. In this case, the only information available to the solver are 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 one another, allowing for the parallel execution of the resulting algorithm. Additionally, as we point out in the chapter, 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.

Then, in Chapter 7, we change our perspective and focus exclusively on LTL A/G contracts. Here we tighten the assumptions on the refinement verification process, that now is required to return, when the refinement does not hold, counterexample traces over the contract variables in addition to the usual yes/no answer. The result is a procedure based on a specialization of OGIS called CounterExample-Guided Inductive Synthesis (CEGIS) [81, 49, 50] 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 an infinite input space, and discuss several performance improvement techniques.

Finally, in Chapter 8, we address the problem of increasing the scalability of the synthesis techniques we developed in the previous chapters. 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 techniques that we introduced to several case studies, including the EPS synthesis problem.

Part of the software and experiments discussed in this dissertation have been implemented in a tool called PYCO[1].

1.2 Main Contributions

The focus of this dissertation is the following:

System design automation can be fully achieved only through the definition and application of rigorous analysis and synthesis techniques. In this dissertation we study, propose, and validate such techniques and algorithms enabling platform-based verification and synthesis of refinements of linear temporal logic specifications.

The main theoretical results in this dissertation are related to the problem of synthesis of contract compositions. We provide, in Chapter 3, a definition of LTL A/G contracts that supports disjoint sets of variables and describe the mechanisms necessary to interconnect them. This adds flexibility when indicating compositions and mappings between specification and library components. In Chapter 6, we provide a general formulation of the CSCL problem, and analyze its complexity. We provide two synthesis approaches based the OGIS paradigm justified by different capabilities of the verification engine(Chapters 6 and 7). We also introduce a novel concept of contract decomposition, in Chapter 8, based on projections, and show how it can be used in the context of the CSCL problem, where synthesis can be broken down into several simpler tasks, guaranteeing the same solution space of the original problem.

We propose, and describe algorithmically, a number of techniques related to verification and synthesis of compositions of contracts. We show how to leverage relations in a library to speed-up refinement verification, and apply it to the EPS case study (Chapter 5). We develop and implement algorithms for all the synthesis techniques we discuss, and evaluate them in several case studies.

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

1.3 Dissertation Outline

The rest of the dissertation is organized as follows.

In Chapter 2, we survey the related literature and discuss its relation with our work. In Chapter 3, then, we formalize the core concepts that will support the rest of the work. 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, and discuss details on techniques and tools to perform basic operations, such as refinement check, which becomes a model checking problem.

In Chapter 4, we introduce the main case study, i.e., the design of a controller for an aircraft electrical power distribution system. We describe the system and then formalize the controller specification as an LTL A/G contract. We also illustrate the contract library that will be used as a reference for both the verification and the synthesis tasks.

In Chapter 5, we present the results about the scalable refinement check technique that leverage library information to build abstractions of compositions. This work is based on a collaboration with Pierluigi Nuzzo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli [41].

In Chapter 6, we introduce the problem of constrained synthesis from component libraries. After analyzing it, we provide a solution based on the OGIS approach which relies on minimal information from the verification engine, i.e., only a yes/no answer. This work has been developed jointly with Stavros Tripakis and Alberto Sangiovanni-Vincentelli [43, 44], while one of the examples has been developed with Richard Lin and Rohit Ramesh.

In Chapter 7, we specialize the CSCL problem and its solution to handle exclusively LTL A/G contracts, requiring the verification engine to returns counterexamples over the contract variables. The solution is a CEGIS algorithm which encodes the counterexamples as state machines. This work is novel. One of the examples has been developed with Íñigo Íncer Romeo.

In Chapter 8, we discuss how to improve the scalability of the synthesis algorithms described in the previous chapters by decomposing specifications, introducing the notion of projection for LTL A/G contracts. This is a joint work with Stavros Tripakis and Alberto Sangiovanni-Vincentelli [42].

Finally, in Chapter 9, we draw conclusions on the work presented in the dissertation and discuss future research directions.

1.4 Related Publications

The material discussed in this dissertation extends the results reported in the following publications.

- [41] A. Iannopollo et al. “Library-based scalable refinement checking for contract-based design”. In: *Design, Automation and Test in Europe Conference and Exhibition*

(DATE), 2014. Mar. 2014, pp. 1–6. DOI: `10.7873/DATE.2014.167`.

- [43] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained Synthesis from Component Libraries". In: *Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besanccon, France, October 19-21, 2016, Revised Selected Papers*. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Springer International Publishing, 2017, pp. 92–110. ISBN: 978-3-319-57666-4. DOI: `10.1007/978-3-319-57666-4_7`.

- [44] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained synthesis from component libraries". In: *Science of Computer Programming* 171 (2019), pp. 21–41. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2018.10.003`.

- [42] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli. "Specification decomposition for synthesis from libraries of LTL Assume/Guarantee contracts". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 1574–1579. DOI: `10.23919/DATE.2018.8342266`.

The work in this dissertation and related publications has been supported by the National Science Foundation (NSF), via the project "ExCAPE: Expeditions in Computer Augmented Program Engineering" (CCF-1139138), 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.

Chapter 2

Related Works

Platform-Based Design Platform-Based Design (PBD) [72, 73, 77, 74, 64] is an iterative design methodology which has been successfully applied in a number of domains, including electronic, automotive, and building automation design. It has been introduced in the late 1980s as a design methodology able to support the design process of complex systems.

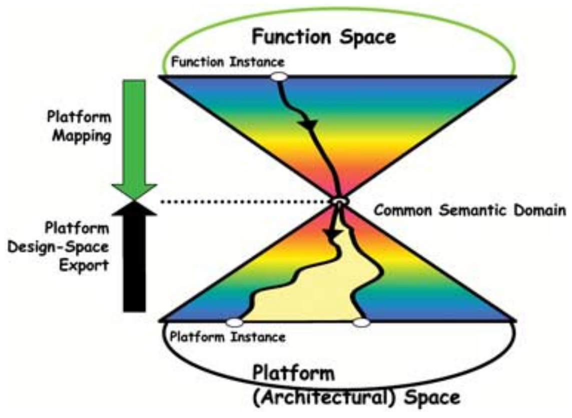


Figure 2.1: A graphical representation of the mapping process typical of Platform-Based Design. A specification, i.e., function, is implemented as a composition of elements in a more concrete abstraction levels through a mapping process. Image from [72].

In PBD, design is carried out as the mapping of a user defined function to a platform instance, as illustrated in Figure 2.1. This platform instance represents a network of interconnected components, chosen from a library. Together with their functionality, in PBD components expose also other characteristics such as composition rules, performance indices, and cost. This additional information is used to optimize the mapping process, according to both functional and non-functional specifications. Two main principles are defined within PBD, abstraction/refinement and composition/decomposition. The former enables a vertical process which allows the design to proceed within different levels of abstraction, while the latter is an horizontal process with describe how design components can be combined or decomposed at the same abstraction level.

Contract Refinement Verification The problem of building and verifying compositions of formally define components has been studied extensively. Pinto *et al.* [64] propose the Contract-Based Specification Language for Platforms (CSL4P). CSL4P 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 [33] describe the inspirational 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 [21, 35]. In Chapter 5, 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 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 is inspired by the platform-based design paradigm [72], where a design at each abstraction layer is also regarded as a platform instance, i.e. a legal 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 this chapter, such relations include refinement rules between contracts in the library. Cimatti and Tonetta [17] 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 Our work on synthesis from component libraries is inspired by two major contributions in this field:

- Pnueli and Rosner [66] show that the problem of synthesizing a set of distributed finite-state controllers such that their network, which is given and fixed, satisfies a given specification is undecidable. In that work, each component in the network is controlled by a finite-state machine (or *program*), and the goal is to synthesize programs that cooperate to satisfy a certain linear temporal logic formula ϕ;
- Lustig and Vardi [54] show that the problem of synthesis from component libraries for *data-flow* compositions is also undecidable. Here components are *transducers*, i.e., finite-state machines able to map a set of input strings to a set of output strings. The specification, also in this case, is a linear temporal logic formula ϕ. In data-flow compositions, the output of a component is fed to another one, while all the components

work synchronously to satisfy the specification. The paper also analyzes another type of composition, *control-flow*, where each component takes full control of the system for a certain period of time, before releasing the system resources to the other components.

Thus, [66] shows that fixing the topology of the network while letting the synthesis process find the components is undecidable, while [54] shows that fixing the components while letting the synthesis process find the topology (possibly by replicating components) is also undecidable. In our work, in Chapters 6 and 7, we too focus on data-flow compositions, and the undecidability results in [66] and [54] are relevant. In our synthesis approach, however, we achieve decidability by imposing a bound on the total number of component instances, which are chosen from a library, positioning our efforts in between the two approaches presented above.

The general idea of synthesis from component libraries adopted here is reminiscent of the work in [48, 34]. There, Jha et al. considered the problem of synthesis of finite loop-free programs from libraries of atomic program statements, using a SMT solver to carry on synthesis. Our work is different, however, as our components are defined as LTL A/G contracts, thus, their have temporal constraints.

A different perspective in synthesis from component libraries has also been described by Alur *et al.* [4]. There, a controller is built out of library components in a control-flow fashion (using the terminology introduced in [54] and discussed above). That approach, while being relevant in the broader topic of synthesis from component libraries, is orthogonal to ours since we focus on data-flow compositions.

Relevant is also the extensive work done in the area of Supervisory Control Synthesis (SCS) [69]. SCS is the problem of synthesizing a controller for a discrete event system, i.e., an automaton, which exposes some controllable and uncontrollable behaviors. The specification defines which behaviors are admissible, and the goal of the controller is to restrict the controllable behaviors of the discrete event system in a way that satisfies the specification. Existing SCS algorithms, however, do not deal with libraries of components but, instead, their goal is to synthesize a controller *ex novo*. Here, we provide a more generic notion of components and focus our effort in synthesizing a controller through the composition of existing components.

Ramesh *et al.* [70] focus on the problem of synthesis of embedded designs from component libraries. In that work, components are represented exclusively by their interface and connections are made on the basis of static relations between component ports. Given a specification, a particularly rich type system takes care of efficiently pruning the search space by solving a constraint satisfaction problem. Although our type system is not as expressive, our approach is more general as we consider components described by more complex specifications (not necessarily static) in addition to their interface.

Manolios *et al.* [55] present a constraint solving framework based on Integer Linear Programming (ILP) where some variables need to be interpreted according to some first-order theories. They develop a constraint solver, Inez, and use it to synthesize architectural models related to the aerospace industry.

Oracle-Guided and Counterexample-Guided Inductive Synthesis, and the combination of Induction, Deduction, and Structure OGIS [49, 48] is a general paradigm to address formal inductive synthesis. It is characterized by the interaction between an inductive learning engine, also called "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 [81, 49, 50], a specialization of OGIS, is a well-known synthesis paradigm which originates from techniques of debugging using counterexamples [78] and *CounterExample-Guided Abstraction Refinement* (CEGAR) [20]. 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 which are processed by an oracle and either declared valid, in which case the algorithm terminates, or 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 [34, 81], and synthesis and completion of distributed protocols [6, 5, 3]. Recently, Seshia [76] 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 [30] studies the problem of specification decomposition into A/G contracts, when they are described in a fragment of Temporal Logic of Actions (TLA$^+$) [51], by proving that unnecessary variables can be efficiently hidden and eliminated from the resulting specifications.

Henzinger *et al.* [37] 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.

Dallal and Tabuada [22], 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. Our goal is similar: breaking down the synthesis process in simpler sub-tasks.

Íncer Romeo *et al.* [46] 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.

Chapter 3

Preliminaries

In this chapter, we provide some basic notions that will be the foundation of the work developed in the other chapters. Here, our goal is two-fold. On the one hand, we describe the details of the formalisms we use and how they relate to each other. For instance, we introduce Linear Temporal Logic (LTL), and discuss how it can be used to specify Assume/Guarantee (A/G) contracts. On the other hand, we provide insights on the techniques we use to manipulate such formalisms, which then we assume in the next chapters. This is the case for LTL manipulation using a model-checker. Here we describe how we can use an off-the-shelf tool to perform the basic operations of validity and satisfiability checks, which will be useful later.

The chapter is organized as follows. Section 3.1 introduces LTL. In Section 3.2, we describe the general A/G contract framework we use in this dissertation, and introduce the concept of contract library in Section 3.3. Then, in Section 3.4, we discuss how to use LTL as a concrete formalism for A/G contracts and show how to practically verify refinement as a model checking problem in 3.5.

3.1 Linear Temporal Logic

Temporal Logic is an extension of propositional logic, introduced by Amir Pnueli [68], 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 example 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. [52]

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 a infinite sequence of reactions:

$$\sigma \in (D^{\Sigma})^{\omega} = r_0, r_1, r_2, \cdots \tag{3.1}$$

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

$$P \subseteq \mathcal{T} \tag{3.2}$$

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:

$$\varphi := \mathit{True} \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \bigcirc\varphi \mid \varphi\, \mathcal{U}\, \psi \tag{3.3}$$

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 follow:

- $\sigma_i \models p$ 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.[33, 67, 71, 16, 65] 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) [15] to encode the whole state space and perform model checking. Explicit state algorithms [40], 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. These algorithms cannot represent a system evolution of infinite length, thus are used to compute bounded model checking (BMC) [13], where a property is verified up to a finite number of temporal steps. While symbolic methods are complete, they tend to use a significant amount of memory to represent the whole system. BMC algorithms, instead, represent a faster solution for a large number of applications, where the verification of properties within a finite horizon is acceptable.

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 players game, where a system implementation is required to be able to react to any possible move from an adversarial environment [67, 71, 27]. If a winning strategy does not exist, then synthesis fails. Otherwise, an implementation of such 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 have discussed how LTL formulas can express sets of infinite-length behaviors. Indeed, Wolper *et al.* [85] show that the models satisfying an LTL formula can be described as a 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 [80].

There are several options to solve the LTL validity problem for a formula φ. One of them is to use a tool which is able to compute an automaton which 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 [32].

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 $\neg\varphi$.

Realizability of an LTL formula, instead, is seen as a game played by two players and it was studied initially by Pnueli and Rosner in the context of LTL synthesis [67]. 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 to 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 [71], but there are several tools that are able to compute it implementing different strategies. For instance, we mention the design tools Ratsy [14] and Tulip [28], or the LTL synthesis tools Acacia+ [26, 27]. 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 [75, 10, 11, 29, 30, 62], 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 [57]. This approach to software engineering has been derived, in turn, by the early work of Floyd and Hoare [31, 39]. The shift of the use of contracts towards system design, however, has been influenced by the early work on interface theories [38, 2, 1, 24].

The concept of contract nicely embrace 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 encapsulate many concrete theories developed over the years for concrete applications. In this thesis, 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 [11].

In our work, the description of a design unit, or just component in the remainder of this chapter, follows Benveniste's definition in [11], which is, in turn, inspired by the Tagged Signal Model, developed by Lee and Sangiovanni-Vincentelli [53]. 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 and it is characterized by an assertion P , i.e., a set of behaviors which 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

$$P_{\|} = \{\sigma \mid \sigma \in P_1 \text{ and } \sigma \in P_2\} = P_1 \cap P_2 \tag{3.4}$$

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

$$M_{\|} = M_1 \parallel M_2 = (\Sigma_P, P_1 \cap P_2) = (\Sigma_P, P_{\|}) \tag{3.5}$$

In the following chapters, we will also use the term component to indicate a generic element in a set or library. To avoid ambiguity, outside of the context of this chapter, 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 thesis, we specialize the description of A/G contracts from [75, 10] to explicitly handle input and output variables.

Definition 1. *An A/G contract is a tuple $C = (I, O, A, G)$ where $I \subseteq \Sigma$ is a set of input variables, $O \subseteq \Sigma$ is a set of output variables, and Σ is the contract alphabet, which is assumed 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 which can and cannot be controlled by the contract[1]. Having such a clear partition is extremely

[1]In [75, 10], 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.

useful, as it allows 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 $M \subseteq M_C$.

3.2.1.1 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

$$M_{C_1} \cap A = (G_1 \cup \overline{A}) \cap A = (G_2 \cup \overline{A}) \cap A \tag{3.6}$$

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

$$G_1 \bigtriangleup G_2 \subseteq \overline{A} \tag{3.7}$$

where $\bigtriangleup$ is the symmetric difference between two sets. If it were, then Equation 3.6 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 3.7 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.

3.2.1.2 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 are 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 of the 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 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.

3.2.1.3 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:

$$\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C} \tag{3.8}$$

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 $C = (I, O, A, G) = \otimes(C_1, C_2)$—also expressed using the infix notation $C = C_1 \otimes C_2$—is defined as follows:

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \tag{3.9a}$$
$$O = O_1 \cup O_2 \tag{3.9b}$$
$$A = (A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)} \tag{3.9c}$$
$$G = G_1 \cap G_2 \tag{3.9d}$$

In principle, a contract representing the composition of two simpler ones should guarantee behaviors which 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 of the inputs of another contract that it is being composed with. The composition, indeed, should consider that some of the 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 new 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 3.9a and 3.9b. 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 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 associative and commutative Given two contracts C_1, C_2, it is immediate to realize that computing their composition according to Equation 3.9 yields the same result, thus $C_1 \otimes C_2 = C_2 \otimes C_1$.

Let us now consider another contract C_3. To show associativity, we need to show that $C_\otimes = (C_1 \otimes C_2) \otimes C_3 = C_1 \otimes (C_2 \otimes C_3) = (I_\otimes, O_\otimes, A_\otimes, G_\otimes)$. It is obvious that computing Equations 3.9b and 3.9d yields the same result independently of the order in which we consider the contracts, obtaining in both cases $O_\otimes = O_1 \cup O_2 \cup O_3$ and $G_\otimes = G_1 \cap G_2 \cap G_3$, respectively. Computing the set of input variables $I_\otimes$ in both cases, we can prove that:

$$I_{(C_1 \otimes C_2) \otimes C_3} = I_{C_1 \otimes (C_2 \otimes C_3)}$$

$$\begin{aligned}&[(I_1 \cup I_2) \setminus (O_1 \cup O_2) \cup I_3] \setminus (O_1 \cup O_2 \cup O_3) = \\ &[I_1 \cup (I_2 \cup I_3) \setminus (O_2 \cup O_3)] \setminus (O_1 \cup O_2 \cup O_3)\end{aligned}$$

$$\begin{aligned}&[(I_1 \cup I_2) \cap \overline{(O_1 \cup O_2)} \cup I_3] \cap \overline{(O_1 \cup O_2 \cup O_3)} = \\ &[I_1 \cup (I_2 \cup I_3) \cap \overline{(O_2 \cup O_3)}] \cap \overline{(O_1 \cup O_2 \cup O_3)}\end{aligned}$$

$$\begin{aligned}&[(I_1 \cup I_2) \cap (\overline{O_1} \cap \overline{O_2}) \cup I_3] \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3}) = \\ &[I_1 \cup (I_2 \cup I_3) \cap (\overline{O_2} \cap \overline{O_3})] \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3})\end{aligned}$$

$$[I_1 \cap \overline{O_1} \cap \overline{O_2} \cup I_2 \cap \overline{O_1} \cap \overline{O_2} \cup I_3] \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3}) =$$

$$[I_1 \cup I_2 \cap \overline{O_2} \cap \overline{O_3} \cup I_3 \cap \overline{O_2} \cap \overline{O_3}] \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3})$$

$$\begin{aligned}&I_1 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3} \cup I_2 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3} \cup I_3 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3} = \\ &I_1 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3} \cup I_2 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3} \cup I_3 \cap \overline{O_1} \cap \overline{O_2} \cap \overline{O_3}\end{aligned}$$

$$\begin{aligned}&(I_1 \cup I_2 \cup I_3) \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3}) = \\ &(I_1 \cup I_2 \cup I_3) \cap (\overline{O_1} \cap \overline{O_2} \cap \overline{O_3})\end{aligned}$$

which is indeed verified and yields $I_\otimes = (I_1 \cup I_2 \cup I_3) \setminus (O_1 \cup O_2 \cup O_3)$.

Finally, we can show that $A_{(C_1 \otimes C_2) \otimes C_3} = A_{C_1 \otimes (C_2 \otimes C_3)}$:

$$\begin{aligned}&\{[(A_1 \cap A_2) \cup \overline{(G_1 \cap G_2)}] \cap A_3\} \cup \overline{(G_1 \cap G_2 \cap G_3)} = \\ &\{A_1 \cap [(A_2 \cap A_3) \cup \overline{(G_2 \cap G_3)}]\} \cup \overline{(G_1 \cap G_2 \cap G_3)}\end{aligned}$$

$$\begin{aligned}&[(A_1 \cap A_2 \cap A_3) \cup A_3 \cap \overline{(G_1 \cap G_2)}] \cup \overline{(G_1 \cap G_2 \cap G_3)} = \\ &[(A_1 \cap A_2 \cap A_3) \cup A_1 \cap \overline{(G_2 \cap G_3)}] \cup \overline{(G_1 \cap G_2 \cap G_3)}\end{aligned}$$

At this point we notice that $\overline{(G_1 \cap G_2)} \subseteq \overline{(G_1 \cap G_2 \cap G_3)}$ and $\overline{(G_2 \cap G_3)} \subseteq \overline{(G_1 \cap G_2 \cap G_3)}$. Thus, by applying the set absorption law, we obtain

$$\begin{aligned}&(A_1 \cap A_2 \cap A_3) \cup \overline{(G_1 \cap G_2 \cap G_3)} = \\ &(A_1 \cap A_2 \cap A_3) \cup \overline{(G_1 \cap G_2 \cap G_3)}\end{aligned}$$

which is indeed, on both sides, $A_\otimes$. Thus, we can generalize composition to an arbitrary number of contracts. For instance, for the composition C of contracts C_1 to C_3, we will simply write $C = C_1 \otimes C_2 \otimes C_3$.

3.2.1.4 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 it 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

$$I' \subseteq I \tag{3.10a}$$
$$O' \supseteq O \tag{3.10b}$$
$$A' \supseteq A \tag{3.10c}$$
$$G' \subseteq G \tag{3.10d}$$

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

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

3.2.1.5 Contract Obligations

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

$$B_C = A \cap G \tag{3.11}$$

Intuitively, a contract obligation describes what are 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 $C_\otimes = C_1 \otimes C_2$ and $C_\otimes' = C_1' \otimes C_2$. Then, $C_\otimes'$ conforms to $C_\otimes$, as we have:

$$(A_1' \cap A_2 \cup \overline{G_1' \cap G_2}) \cap (G_1' \cap G_2) \subseteq (A_1 \cap A_2 \cup \overline{G_1 \cap G_2}) \cap (G_1 \cap G_2) \tag{3.12}$$
$$(A_1' \cap A_2) \cap (G_1' \cap G_2) \subseteq (A_1 \cap A_2) \cap (G_1 \cap G_2) \tag{3.13}$$
$$(A_1' \cap G_1') \cap (A_2 \cap G_2) \subseteq (A_1 \cap G_1) \cap (A_2 \cap G_2) \tag{3.14}$$
$$B_{C_1'} \cap B_{C_2} \subseteq B_{C_1} \cap B_{C_2} \tag{3.15}$$
$$B_{C_\otimes'} \subseteq B_{C_\otimes} \tag{3.16}$$

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

3.2.1.6 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 by the defintion by de Alfaro and Henzinger in [2], 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.

$$\bar{\theta}(x) = \begin{cases} y & \text{if } (x, y) \in \theta \\ x & \text{otherwise} \end{cases} \tag{3.17}$$

A connection $\vartheta : \mathbb{C} \times \Theta \to \mathbb{C}$, where Θ is the set of all interconnects, is a function which maps a contract and an interconnect to a new contract. Given a contract $C = (I, O, A, G)$ and an interconnect θ, we indicated 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

$$I_\theta = (I \cup \{\ y \quad \text{if } \exists y\colon (x,y) \in \theta \ \big|\, x \in I\}) \setminus O_\theta \tag{3.18a}$$

$$O_\theta = O \cup \{\ x \quad \text{if } \exists y\colon (x,y) \in \theta \ \big|\, x \in I\} \cup \{\ y \quad \text{if } \exists y\colon (x,y) \in \theta \ \big|\, x \in O\} \tag{3.18b}$$

$$A_\theta = A \cup \overline{\rho_\theta} \tag{3.18c}$$

$$G_\theta = G \cap \rho_\theta \tag{3.18d}$$

and

$$\rho_\theta = \bigcap_{x,y \in I_\theta \cup O_\theta, (x,y) \in \theta} \{\sigma \mid x \equiv y\} \tag{3.19}$$

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 $C\theta = C \otimes C_\theta$, 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 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 chapters.

Definition 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 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

$$L = (\mathcal{Z}, \mathcal{R}) \tag{3.20}$$

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 constraints over 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 composition are legal. 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 chapters. 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 illegal 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 $\psi \leftrightarrow (\psi \vee \neg\varphi) \Rightarrow \psi \to \varphi$, where $\to$ is the symbol for material implication and $\leftrightarrow$ is double material implication.

The composition of two contracts $C_\otimes = C_1 \otimes C_2$ is computed as:

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2) \tag{3.21a}$$

$$O = O_1 \cup O_2 \tag{3.21b}$$

$$\varphi_\otimes = (\psi_1 \wedge \psi_2) \rightarrow (\varphi_1 \wedge \varphi_2) \tag{3.21c}$$

$$\psi_\otimes = \psi_1 \wedge \psi_2 \tag{3.21d}$$

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:

$$I' \subseteq I \tag{3.22a}$$

$$O' \supseteq O \tag{3.22b}$$

$$\varphi \rightarrow \varphi' \text{ is valid} \tag{3.22c}$$

$$\psi' \rightarrow \psi \text{ is valid} \tag{3.22d}$$

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, as discussed in Sections 3.1.4 and 3.5.

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 3.18a and 3.18b, and:

$$\varphi_\theta = \rho_\theta \rightarrow \varphi \tag{3.23a}$$

$$\psi_\theta = \psi \wedge \rho_\theta \tag{3.23b}$$

where we can cast the set of all the traces in which all the pairs in θ are equivalent, introduced in Equation 3.19, as:

$$\rho_\theta = \bigwedge_{x,y \in I_\theta \cup O_\theta, (x,y) \in \theta} \Box(x = y) \tag{3.24}$$

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's 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 which can handle all the outputs of the system implementing the contract. In the experiments performed in the context of this thesis, all the contracts in the libraries have been verified for consistency and compatibility using Ratsy [14].

3.5 LTL Satisfiability and Validity as a Model Checking Problem

The model checking problem is 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 $\mathcal{L}(M) \cap \mathcal{L}(\neg\phi) = \emptyset$ [19]. 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 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 [56], and discuss how we can use a model checker supporting that language, i.e., NuXMV [16], to check for satisfiability and validity of a certain LTL formula.

3.5.1 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.

The listing in Figure 3.1, which models a 3-bit counter, shows the typical structure of an SMV program. Each module defines a set of variables which 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.

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

```
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 3.1: An SMV program implementing a 3-bit counter[2].

Each module can define what is the legal 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 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 which 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 [16] for a full description of the SMV language.

3.5.1.1 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 legal. 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 to 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 2 (SMV Program for LTL Validity Check)**.** *Let $\phi = \Box(a \wedge b) \rightarrow \Diamond a$ be a 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 3.2, which is obviously true.*

```
MODULE foo
VAR a : boolean;
    b : boolean;

MODULE main
VAR m : foo;

LTLSPEC G (m.a & m.b) -> F m.a
```

Figure 3.2: The SMV program for checking the validity of $\phi = \Box(a \wedge b) \rightarrow \Diamond a$.

3.5.2 Structure of a NuXMV Counterexample

The listing in Figure 3.3 shows the structure of a counterexample generated by NuXMV. In this case, we model checked the program in Figure 3.1 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 3.1.

Counterexamples can either be finite or infinite. If they are infinite, they will have a lazo-shape 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 3.3 (e.g., see line 5).

```
 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 3.3: Counterexample generated by model-checking the program in Figure 3.1 with the specification `G bit2.carry_out`. The dots in Line 16 have been added manually to indicate the shortening of the trace.

Chapter 4

The Aircraft Electrical Power System Case Study

The design of aircrafts and aircraft parts has been, arguably, one of the the most challenging and advanced applications of system engineering of the last century, and today they are certainly among the most complex cyber-physical systems. In such vehicles, a number of completely different subsystems, i.e., hydraulic, pneumatic, electric, etc., are designed to flawlessly cooperate together under very tight design constraints [59].

With the advent of the so called “more electric airacraft” [79], in recent years, the field of aircraft design has been completely revolutionized. The Boeing 787 Dreamliner, which has started flying commercially in 2014, is the most descriptive example of such deep paradigm shift. The reasons behind such transformation are numerous, including reduced fuel consumption, reduced maintenance costs, and improved reliability. Tasks that were traditionally being assigned to the pneumatic system, such as the cabin pressurization, are executed using electricity-driven compressors, eliminating an expensive system of ducts, valves, and temperature and pressure controls that transported compressed air from the engines to the rest of the plane. To support these new responsibilities, the EPS in modern aircrafts has been completely upgraded, moving from a centralized to a distributed architecture, as shown in Figure 4.1.

A redundant, distributed architecture, however, presents new challenges for the EPS control software. Dealing with a complex network of components introduces new requirement and risks that need to be carefully addressed. Formal languages can be used to describe specifications for an EPS avoiding ambiguities and enabling formal analysis, at the price of reduced scalability. Throughout this thesis, we will use the problem of designing the EPS control software as a unifying case study. Specifically, our goal is to develop techniques to verify and synthesize a controller unit for the EPS architecture, which are more scalable than traditional methods. We only focus on functional aspects of the design, which can be described using LTL. We assume that a pre-existing library of LTL A/G contracts describing components, i.e., controllers for subsets of the EPS plant, described later in this chapter, is available. The methodology and algorithms we propose are meant to address the problem of

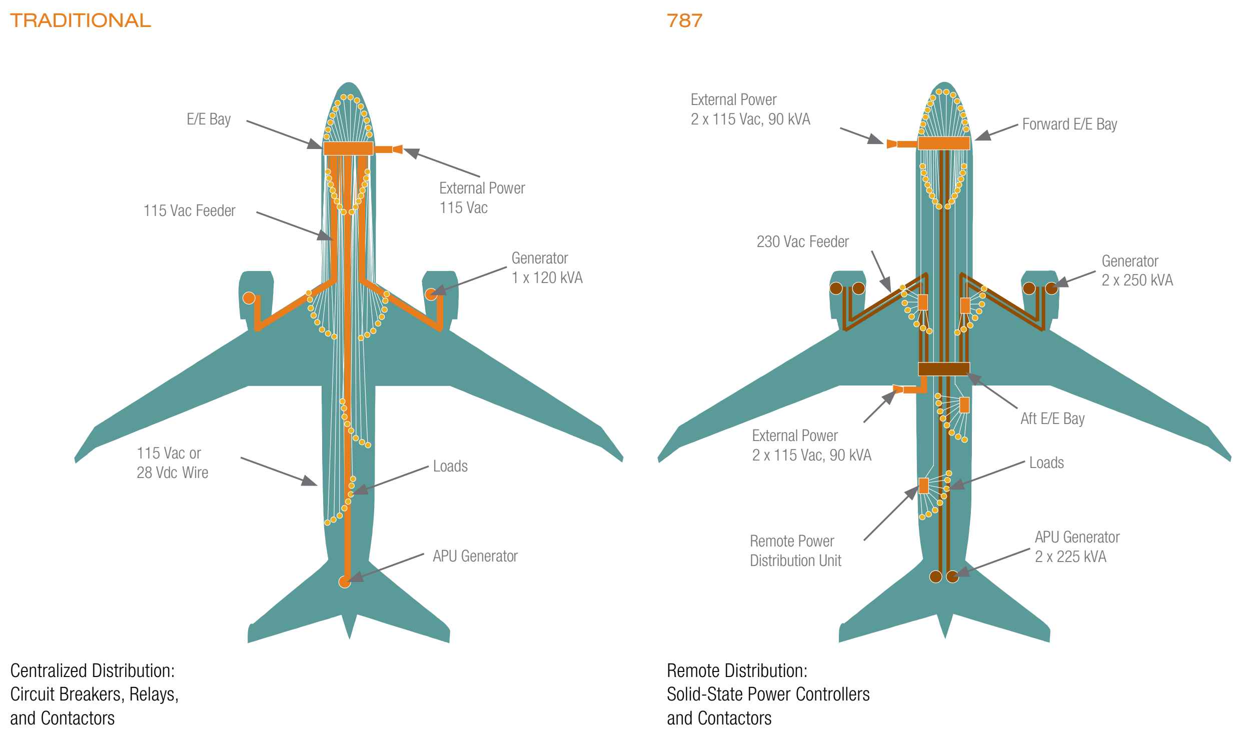


Figure 4.1: A comparison between the EPS system in traditional aircrafts and the one typical of modern aircrafts, such as the Boeing 787. The main difference is the distributed architecture of the modern design, introduced to increase reliability and reduce cost. The details about the components in the picture are discussed in Section 4.1. Image from [79].

how such components can be combined to obtain a full controller for the plant.

4.1 EPS Details

Figure 4.2 shows an EPS architecture patented by Honeywell International Inc. [58], which implement the changes discussed in the previous section. Figure 4.3 presents a simplified view of the same architecture, in the form of a *single-line diagram*[1] [59, 61, 41]. Generators (as those on the top left and right sides of the diagram) deliver power to the loads (e.g., avionics, lighting, heating, and motors) via AC and DC buses. In the event of generator failures, *Auxiliary Power Units* (APUs) will provide the required power. Some buses supply loads which are critical, therefore they cannot be unpowered for more than a predefined amount of time. Other, non-essential, buses supply loads that may be shed in the case of a fault. The power flow from sources to loads is determined by contactors, which are electromechanical

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

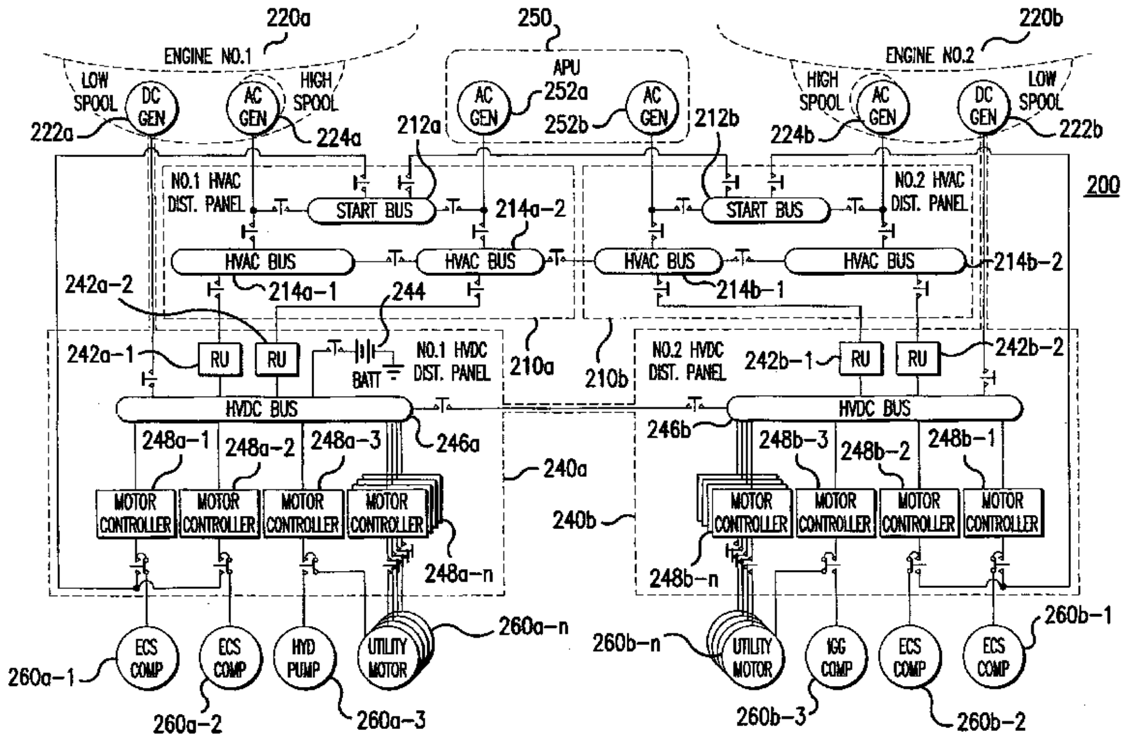


Figure 4.2: Single line diagram of a typical EPS, as it appears in [58]. The system includes redundant high-voltage (AC) and a low-voltage (DC) power distribution sections. Generators and loads are connected through buses. A set of contactors controls the power flow.

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 changes in system conditions or failures and reroute power by actuating the contactors, ensuring that essential buses are adequately powered. Generators, APUs, and TRUs are components subject to failures.

4.2 EPS Specification

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

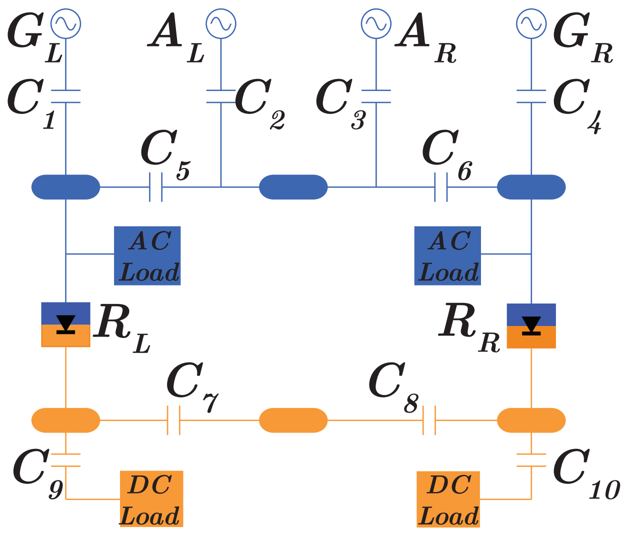


Figure 4.3: Simplified single line diagram of the EPS [58]

.

Table 4.1 illustrates the set of specifications that the BPCU needs to satisfy. The first two rows on the left describe what are the input and output ports of the EPS plant and their types, indicated in parenthesis next to the port names (Figure 4.4 shows the type tree associated with ports in the specification and library components). Types compatibility is encoded as a set of library constraints, described in Section 4.3. In total, each specification is defined over 6 input and 10 output ports.

Input ports $G_L, G_R, A_L, A_R, R_L, R_R$ represent the environment event of failure of the left and right generator, APU, and TRU, respectively. Output ports $C_1, \dots, C_{10}$ represent the

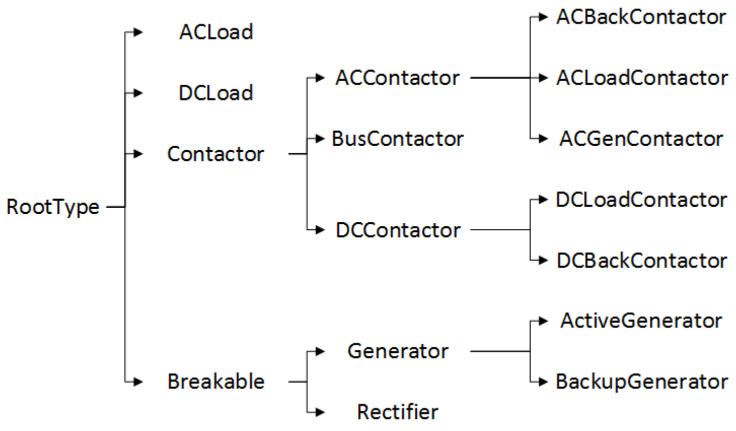


Figure 4.4: Tree representing the typeset used in the EPS case study.

state of the contactors. The second column of Table 4.1 describes a set of 9 specifications, all

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

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

Table 4.1: Set of system specifications $S_1 \ldots S_9$ 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.

sharing the same assumptions. In this example, we assume from the environment that all the components do not start to operate in a faulty state (see, for instance, $\neg G_L$ in the first line of the assumptions in Table 4.1, referring to the left generator), and if a component breaks, then it will stay broken (specified, for the left generator, by $\Box(G_L \rightarrow \bigcirc G_L)$). Specifications S_1 to S_4 require that if a generator or APU breaks, then it will be disconnected from the rest of the EPS in the next execution step. Note that S_1 and S_2 require also the two generators to be initially connected to the rest of the plant. S_5 requires the absence of a short circuit between the two APUs, while S_6 requires the absence of a short circuit between generators in case they are both healthy (after an initial setup period). Furthermore, S_7 specifies that bus B_3 needs to be isolated if no faults in generators or APUs occur. Finally, S_8 and S_9 require that DC loads need to be connected to the plant if at least one TRU is working correctly.

4.3 EPS Library

Table 4.2 shows the components and the user-defined constraints (in this example only type compatibility) in the library. Every component is described by its I/O ports (annotated with their types), and its specification as an A/G pair. All the components make some assumptions over the state of a certain type of EPS elements and provide a guarantee over the state of some contactors. Consider, for instance, component B_1. It just assumes that a certain generator is not initially broken (note that the type of the input variable allows it to be connected to either a generator or an APU), and guarantees that the contactor will be always open. Clearly, B_1 is not a good candidate to satisfy either S_1 or S_2, since they require the contactor to be closed at least initially. Similarly, all the other components in the library encode a particular behavior that can be used to control parts of the EPS.

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

Table 4.2: 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.

Chapter 5

More Scalable Refinement Checking with Contract Libraries

In this chapter, we discuss techniques for the verification of designs using LTL A/G contracts. In general, given a specification contract and a system described by a composition of contracts, system verification reduces to checking that the composite contract refines the specification contract, i.e., that any implementation of the composite contract implements the specification contract and is able to operate in any environment admitted by it. For LTL A/G contracts, refinement checking reduces to an LTL satisfiability checking problem, which can be very expensive to solve for large composite contracts. We describe a scalable approach to refinement checking that relies on local refinement assertions stored in a library of contracts. We propose an algorithm that, given such a library, breaks down the refinement checking problem into multiple successive refinement checks, each of smaller scale. We illustrate the benefits of the approach on the EPS case study introduced in Chapter 4, where we are able to obtain up to two orders of magnitude improvement in terms of execution time.

After introducing and discussing the motivation of this work in Section 5.1, in Section 5.2 we present the formulation of the refinement check problem with library (RCPL). Section 5.3 and Section 5.4 detail, respectively, the proposed algorithm and the application example. Finally, we derive conclusions in Section 5.5.

5.1 Introduction

An important task for the successful deployment of a contract-based methodology is refinement checking. In all contract frameworks, given a global specification contract and a system, also described by a composition of contracts, system verification reduces to checking that the composite contract refines the specification contract. This verification step goes beyond the mere, although important, validation of user-provided designs, but is a key element in the definition of highly automated design frameworks, where maintaining consistency between successive design refinement steps is of paramount importance. Even if refinement checking can be carried out compositionally, it can still be very expensive to solve for large composite contracts. For LTL A/G contracts, refinement checking reduces to an LTL satisfiability checking problem, which is PSPACE-complete [80]. Moreover, even if contracts are not captured in LTL but instead are expressed directly in an automata-based formalism such as interface automata, for which refinement checking is polynomial [1], the method still suffers from scalability issues due to state explosion. Indeed, the size of the system automaton is often prohibitive, as the system is formed by composing several sub-systems.

We propose an algorithm which, given such a library, breaks down the refinement checking problem into multiple successive refinement checks, each of smaller scale. While our principal focus here is to illustrate the benefits of our approach when specification and components are expressed as LTL A/G contracts, we want to stress how, in principle, the same considerations hold for any contract framework.

The contribution of our work is twofold:

- we propose an algorithm to improve the performance of refinement checking, the core verification task underlying any proof obligation in contract-based design;
- we illustrate the benefits of a library-based approach for contract-based verification on a case study of industrial relevance.

5.2 Problem formulation

Definition 5 (Refinement Check Problem (RCP))**.** *Let $\mathcal{Z}$ be a set of contracts, and $C_s = (C_1 \otimes C_2 \otimes \cdots \otimes C_n)\theta$ be a composition of contracts specifying a system, where $C_1 = (I_1, O_1, \varphi_1, \psi_1)$, $\ldots, C_n = (I_n, O_n, \varphi_n, \psi_n) \in \mathcal{Z}$ and θ is an interconnect as defined in Section 3.2.1.6. Let also C_p be a property expressed as a contract. Then, to ensure that any implementation of C_s satisfies C_p and can operate in all environments admitted by C_p, we need to verify that $C_s \preceq C_p$.*

For LTL A/G contracts, as described in Section 3.5, RCP can be solved using LTL satisfiability solving techniques, which suffers from the well-known state-explosion problem. In subsequent sections, we will refer to RCP indicating a routine that solves the refinement problem using such techniques. To perform such task more efficiently, we recur to a different problem formulation, which relies on a library of contracts as an additional input.

5.2.1 Library of Contracts and Library Validation Problem

As introduced in Section 3.3, a library of contracts L is a pair $(\mathcal{Z}, \mathcal{R})$ where:

- $\mathcal{Z} = \{C_1, ..., C_n\}$ is a finite set of contracts such that each contract has unique variables;
- $\mathcal{R}$ is a set of relations over the contracts in $\mathcal{Z}$.

In the context of this chapter, we specialize the set of constraints $\mathcal{R}$ to represent refinement relations between contracts in $\mathcal{Z}$. Ideally, refinement relations are assertions made by library designers based on their knowledge of the system architecture at hand.

Every refinement relation $R_i \in \mathcal{R}$ has the form

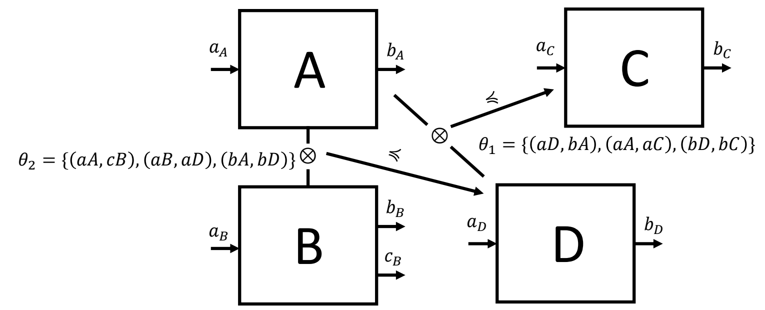
$$R_i = (C_{Ri}, C_{Ai}, \theta_i)$$

where $C_{Ri} = (C_{i_1} \otimes \cdots \otimes C_{i_k})\theta_i$ for a certain interconnect θ_i, $C_{i_1}, ..., C_{i_k}, C_{Ai} \in \mathcal{Z}$. For each relation R_i, we have that

$$R_i = (C_{Ri}, C_{Ai}, \theta_i) \Rightarrow \begin{cases} C_{Ri} \preceq C_{Ai} & \text{if } k > 1 \\ C_{Ri} \prec C_{Ai} & \text{if } k = 1 \end{cases} \tag{5.1}$$

meaning that if $k = 1$, we require that C_{Ri} strictly refines C_{Ai}, i.e., the two contracts are not equivalent, thus $C_{Ri} \preceq C_{Ai}$ and $C_{Ai} \not\preceq C_{Ri}$. This constraint is introduced to avoid, in the library, the presence of circular dependencies, and therefore ensure termination of the algorithms presented below. For sake of simplicity, later, we will call the contract C_{i1} the *root* of R_i.

Definition 6 (Library Validation Problem (LVP))**.** *We say that a library $L = (\mathcal{Z}, \mathcal{R})$ is valid if all its refinement relations in $\mathcal{R}$ are true. The LVP is, then, the problem of checking whether a given library is valid.*



a_A
A
b_A
a_C
C
b_C
$\preccurlyeq$
$\theta_1 = \{(aD, bA), (aA, aC), (bD, bC)\}$
$\theta_2 = \{(aA, cB), (aB, aD), (bA, bD)\}$
$\preccurlyeq$
b_B
a_B
B
c_B
a_D
D
b_D

Figure 5.1: Example contract library with refinement assertions.

Example 3 (Contract Library and Refinement Relations)**.** *Figure 5.1 represents a contract library, L', where its refinement relations are emphasized. In this case, $L' = (\mathcal{Z}', \mathcal{R}')$ where $\mathcal{Z}' = \{A, B, C, D\}$, and $\mathcal{R}' = \{R_1, R_2\}$. We have that:*

- $R_1 = ((A \otimes D)\theta_1, C, \theta_1)$, *where* $\theta_1 = \{(a_D, b_A), (a_A, a_C), (b_D, b_C)\}$*;*
- $R_2 = ((A \otimes B)\theta_2, D, \theta_2)$, *where* $\theta_2 = \{(a_A, c_B), (a_B, a_D), (b_A, b_D)\}$*.*

When a library of contracts defined as in Section 5.2.1 is available as an additional input, the Refinement Check Problem with Library (RCPL) can be formulated, then, as a special case of RCP.

Definition 7 (The Refinement Check Problem with Library)**.** *Let C_p be a contract representing a system specification, $L = (\mathcal{Z}, \mathcal{R})$ be a contract library, and $C_s = (C_1 \otimes C_2 \otimes \cdots \otimes C_n)\theta$ be a system contract, for a certain θ and $C_1, C_2, \ldots, C_n \in \mathcal{Z}$. Then, check whether $C_s \preceq C_p$.*

5.3 Scalable Contract Refinement Checking

5.3.1 Library Validation

Given a library defined as in Section 5.2.1, the library verification process ensures that all its refinement assertions are correct. If any of such refinement relations is not verified, the returned value of the algorithm will be *False*. A description of the library verification process is given in Algorithm 1.

```
1 function LibraryValidation:
      Input: library $L = (\mathcal{Z}, \mathcal{R})$
      Output: $True$ if all refinement relations in the library are true, $False$ otherwise
2     foreach $(C_{Ri}, C_{Ai}, \theta_i) \in \mathcal{R}$ do
3         if $k > 1$ and $C_{Ri} \not\preceq C_{Ai}$ then return $False$;
4         if $k = 1$ and $C_{Ri} \not\prec C_{Ai}$ then return $False$;
5     end
6     return $True$;
7 end
```
Algorithm 1: Checks whether the input library is valid.

Each refinement check in the algorithm is performed by solving an RCP instance as described in Definition 5, which is reasonable in terms of computation time, since aggregations of library contracts are expected to have a small size. Moreover, the overall efficiency of the LVP is deemed to be less critical since it is performed only once, outside of the main verification flow.

5.3.2 Checking Refinement with Contract Libraries

Our refinement checking procedure is described in Algorithms 2, 3, and 4. We start with a valid library $L = (\mathcal{Z}, \mathcal{R})$, a property contract C_p (where possibly $C_p \notin \mathcal{Z}$), and a system contract C_s, obtained as the composition of a set of contracts $\mathcal{S} = \{C_1, \ldots, C_n\}$, $C_1, \ldots, C_n \in \mathcal{Z}$, opportunely interconnected according to a certain θ. The system contract C_s represents the specification of a complex system, while the property contract C_p captures a requirement that must be satisfied by the system. We further assume that, given a variable v such that $v \in O_i$, belonging to a contract $C_i \in \mathcal{S}$, then $v \notin O_j$, for $C_j \in \mathcal{S}$ and $j \neq i$, meaning that each variable is controlled only by one contract in $\mathcal{S}$ or by a legal environment of C_s.

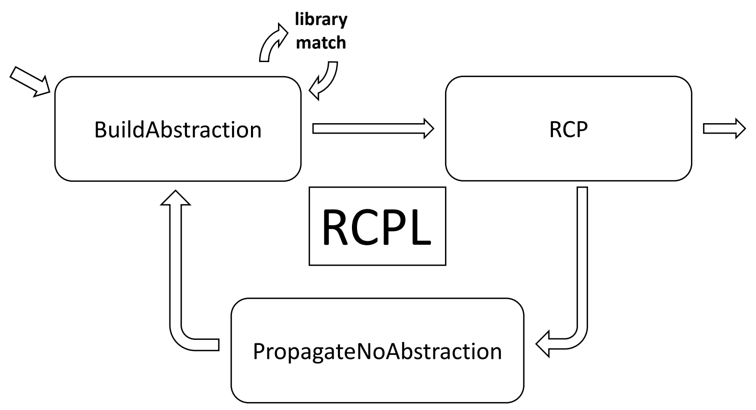


Figure 5.2: Representation of the RCPL algorithm.

We solve the RCPL using the algorithm represented in Figure 5.2 and consisting of two nested loops. In the inner loop, the procedure `BuildAbstraction` tries to create a maximal abstraction for C_s given the refinement assertions in L and an indication about which contracts can be abstracted. As a result, some of the contracts in $\mathcal{S}$ will be replaced by an equal or smaller number of more abstract contracts, resulting in a composition that we will denote as C_{abstr}. Since, in general, a more abstract contract is expressed by smaller formulas, C_{abstr} will be simpler and more compact than C_s. The indication on which contracts can be abstracted is provided via the outer loop by the routine `PropagateNoAbstraction`.

In the outer loop, refinement between C_{abstr} and C_p is checked by the RCP routine. If $C_{abstr} \preceq C_p$ holds, then $C_s \preceq C_p$ will also hold since, by construction, we have $C_s \preceq C_{abstr}$ and the RCPL routine terminates. If the property is not verified at the current level of abstraction, subsequent iterations will use a less and less abstracted representation of C_s. In the worst case, no abstraction is performed and RCPL reduces to an instance of RCP with the not abstract contract. The outer loop of the RCPL procedure is illustrated in Algorithm 2. To control the level of abstraction, each contract (including C_p) has an associated Boolean flag that corresponds to a *no-abstraction* constraint. If the flag is *True*, the contract will not be substituted by a more abstract one, even if this is available in the library. As shown in line 9 in Algorithm 2, the main loop terminates when $C_{abstr} \preceq C_p$ or when the function `BuildAbstraction` cannot return a more abstract contract.

The procedure `BuildAbstraction` in Algorithm 3 implements the inner loop of RCPL. It accepts as inputs a library $L = (\mathcal{Z}, \mathcal{R})$, a contract C_s composed of contracts in $\mathcal{Z}$, and a list of flags A built as described in Algorithm 2. The algorithm tries to abstract C_s by using the information in L until no progress is made. At each iteration, a copy of the current set of contracts $\mathcal{S}$ is maintained in $\mathcal{S}'$ and the abstractionCondition is checked on each contract $C_k \in \mathcal{S} \cap \mathcal{S}'$. If it evaluates to true, a subset of contracts is matched to an aggregation of contracts C_{Ri} in L and then replaced by its abstraction C_{Ai}. The abstractionCondition requires the following sub-conditions to hold:

```
function RCPL:
    Input: specification as contract $C_p$, library of contracts $L = (\mathcal{Z}, \mathcal{R})$, contract
           $C_s = (C_1, \otimes \cdots \otimes C_n)\theta$ where each $C_i \in \mathcal{Z}$
    Output: $True$ if $C_s \preceq C_p$, $False$ otherwise
    $\mathcal{S} \leftarrow \{C_1, \ldots, C_n, C_p\}$;
    $A \leftarrow$ hash table such that $A[C_p] = True$ and $A[C_i] = False, 1 \leq i \leq n$;
    $C_{abstr} \leftarrow$ copy of $C_s$;                // init assignment, copy as in Def. 4
    repeat
        $C_{old} \leftarrow C_{abstr}$;
        $\mathcal{S}' \leftarrow$ BuildAbstraction($A, L, C_s$);        // see Algorithm 3
        $C_{abstr} \leftarrow \otimes\{C_i \mid C_i \in \mathcal{S}'\}\theta$;
        if $C_{abstr} \preceq C_p$ then return $True$;
        $A \leftarrow$ PropagateNoAbstraction($A, \mathcal{S}, \theta$);   // see Algorithm 4
    until $C_{abstr} \not\equiv C_{old}$;
    return $False$;
end
```

Algorithm 2: Solves the RCPL problem by iteratively building and verifying abstractions. In each iteration, the abstraction is closer to the original contract.

- C_k is not flagged by a *no-abstraction* constraint, that is $A[C_k] = False$;
- $\exists R_i = (C_{Ri}, C_{Ai}, \theta_i) \in \mathcal{R}$ such that C_k and the root of R_i are equivalent;
- $\exists C_{k_1} \ldots C_{k_m} \in \mathcal{S} \cap \mathcal{S}'$, such that $A[C_{k_1}] = \cdots = A[C_{k_m}] = False$, and a renaming θ_a such that $C_{Ri}\theta_a = C_k \otimes C_{k_1} \otimes \cdots \otimes C_{k_m}$, i.e., there exists a subset of contract that can be abstracted and such that, when composed with C_k, generate a contract that is equivalent to C_{Ri};
- $(O_{Ri\theta_a} \setminus O_{Ai\theta_a}) \cap I_r = \emptyset,\ \ C_r \in \mathcal{S}' \setminus \{C_k, C_{k_1}, \ldots, \mathcal{Z}_{k_m}\}$, i.e. no substitution is made if there is some other contract in C_s, which is not in $\{C_k, C_{k_1}, \ldots, C_{k_m}\}$, and such that at least one of its input variables is missing a connection in the abstract contract.

The replacement of the contracts in the original list as well as the selection of candidate abstractions from the library are currently performed in a random order. More sophisticated heuristics are possible, although we do not explore their application at this time.

Termination of `BuildAbstraction` is guaranteed since contract C_{abstr} will not change after a certain number of iterations. In fact, the number of matches of contracts in $\mathcal{R}$ performed in line 6 is finite. Therefore, since the library is finite, we just need to prove the absence of circular dependencies between contract relations in $\mathcal{R}$. To show this, we observe that for each matching relation $R_i = (C_{Ri}, C_{Ai}, \theta_i)$, with $C_{Ri} = (C_{i_1} \otimes \cdots \otimes C_{i_k})\theta_i$, there are two possible cases. If $k > 1$, after replacing C_{Ri} with C_{Ai}, the number of contracts in $\mathcal{S}$ decreases. Obviously, this operation can only be performed a finite number of times. On the

```
1  function BuildAbstraction:
       Input: library of contracts $L = (\mathcal{Z}, \mathcal{R})$, composite contract $C_s = (C_1, \otimes \cdots \otimes C_n)\theta$
              where each $C_i \in \mathcal{Z}$, a hash table $A$ as in Algorithm 2
       Output: set of contracts $\mathcal{S} = \{C_{a_1}, \ldots, C_{a_m}\}$, such that $C_s \preceq (C_{a_1} \otimes \cdots \otimes C_{a_m})$
2      $\mathcal{S} \leftarrow \{C_1, \ldots, C_n\}$;
3      repeat
4          $\mathcal{S}' \leftarrow \mathcal{S}$;
5          foreach $C_k \in \mathcal{S} \cap \mathcal{S}'$ do
6              if abstractionCondition is satisfied for some $C_{k1}, \cdots, C_{km}, C_{Ai}$, and $\theta_a$ then
7                  $\mathcal{S}' \leftarrow (\mathcal{S}' \setminus \{C_k, C_{k1}, \cdots, C_{km}\}) \cup \{C_{Ai}\theta_a\}$;
8                  $C_{abstr} \leftarrow \otimes\{C_i \mid C_i \in \mathcal{S}'\}\theta$;
9              end
10         end
11         $\mathcal{S} \leftarrow \mathcal{S}'$;
12         $C_{old} \leftarrow C_{abstr}$;
13     until $C_{abstr} \equiv C_{old}$;
14     return $\mathcal{S}$;
15 end
```

Algorithm 3: Uses relations in the library to build an abstraction of the input contract C_s. Cf. Section 5.3.2 for details on the satisfaction of the condition in line 6.

other hand, if $k = 1$, since we requires $C_{Ri} \prec C_{Ai}$, we will always have $C_{Ai} \not\preceq C_{Ri}$. Therefore, it is impossible to find in the library a relation $R'_i = (C_{Ai}, C_{Ri}, \theta_i)$, which would represent a circular dependency between R_i and R'_i.

The runtime of `BuildAbstraction` is mostly determined by the time it takes to find a matching between a set of library contracts and a subset of the contracts composing C_s. Such a matching problem can be reduced to a graph isomorphism problem, which can be efficiently solved [82, 8]. In our case, graphs can be built to represent contract compositions, while incorporating information on the names of the variables of the component contracts and their isomorphism.

The heuristic used in the propagation of the *no-abstraction* constraint is, finally, detailed in Algorithm 4. We propose an incremental propagation of the constraint according to the syntactical dependence between contracts. The algorithm receives as a parameter the list of contracts that compose C_s and their interconnection θ, extended with the addition of the property contract C_p (the first to receive the *no-abstraction* mark). Each time `PropagateNoAbstraction` is called, the *no-abstraction* mark will be propagated to all contracts that share at least one of their output variables with a marked contract. This approach is similar to the concept of "cone of influence" used in Counterexample-Guided Abstraction Refinement [20].

We provide an example of execution of our algorithm in Figure 5.3. The contract in

```
function PropagateNoAbstraction:
    Input: set of contracts $\mathcal{S} = \{C_1, \ldots, C_n\}$, hash table $A$ as in Algorithm 2,
           interconnect $\theta$
    Output: hash table $A'$
    $A' \leftarrow A$;
    $M \leftarrow \{\}$;
    foreach $C_k \in \mathcal{S}$ do
        if $A'[C_k] = True$ then
            foreach $C_h \in \mathcal{S}$ do
                if $I_k \cap O_h \neq \emptyset$ then
                    $M \leftarrow M \cup \{C_h\}$;
                else if $\theta$ connects variables in $I_k$ and $O_h$ then
                    $M \leftarrow M \cup \{C_h\}$;
                end
            end
        end
    end
    foreach $C_i \in M$ do
        $A'[C_i] \leftarrow True$;
    end
    return $A'$;
end
```

Algorithm 4: Marks a contract as non-abstractable if it is connected to a non-abstractable contract. Each time the function is called, the size of the non-abstractable set grows until all contracts are non-abstractable.

Figure 5.3 (a) is obtained by composition of contracts from the library in Figure 5.1. The arrows denote connections. We assume that the property contract C_p is mapped only to variables a_B and c_D. We then call the RCPL algorithm using C_p, C_s in Figure 5.3 (a), and L' from Example 3. At the first execution of BuildAbstraction, all contracts can be potentially abstracted. However, there are only two possible matches between portions of the architecture in Figure 5.1 and the refinement relations in $\mathcal{R}'$. In particular, the composite contract $B \otimes A_2$ can be abstracted as D_1, an instance of D, while $A_1 \otimes D$ can be abstracted as C. However, $B \otimes A_2$ does not satisfy the last condition for the abstractionCondition to hold in line 6 of Algorithm 3. In fact, replacing $B \otimes A_2$ with D_1 would cause the loss of a variable (b_B) that should be shared with A, hence an incorrect abstraction. Conversely, the substitution of $A_1 \otimes D$ with C is legal and the resulting contract composition, C_{abstr}, is shown in Figure 5.3 (b). If $C_{abstr} \preceq C_p$, the algorithm would terminate by executing an instance of the RCP on a more compact representation of the system contract. Otherwise, if $C_{abstr} \not\preceq C_p$, PropagateNoAbstraction would mark D with a *no-abstraction* annotation. At this

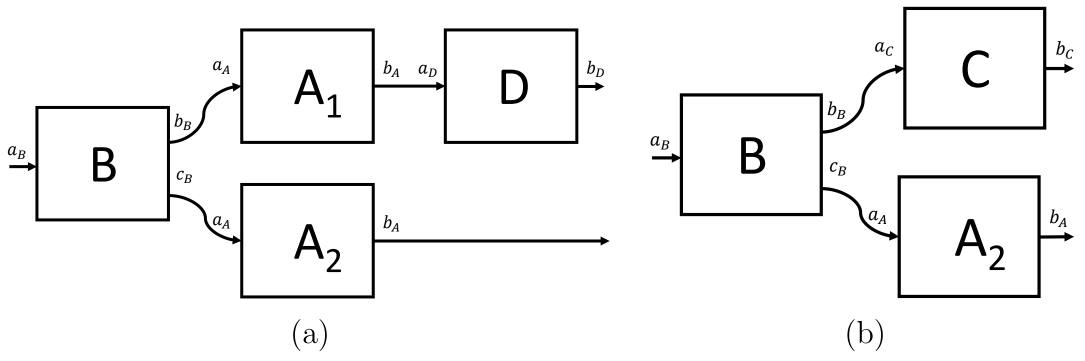


Figure 5.3: Representation of a composite contract obtained from the library in Figure 5.1 (a) and its abstraction (b).

point, no contract aggregation can be further abstracted, and the algorithm terminates by solving an instance of the RCP on the original composition.

5.4 Application Example: the EPS Case Study

The proposed algorithm was implemented in Python and applied on the verification of a controller for the EPS case study defined in Chapter 4. To solve the LTL satisfiability problems, we used NuSMV [18]. All tests were performed on a 2.3-GHz Intel Core i7 machine with 8 GB of RAM.

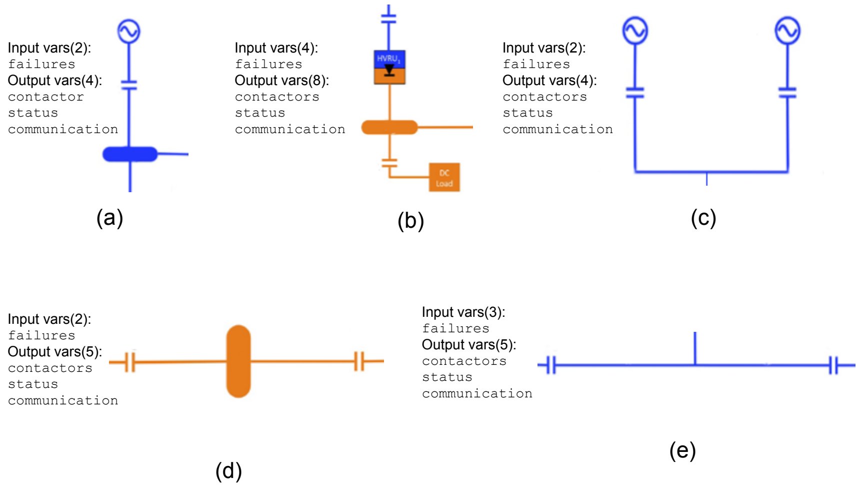


Figure 5.4: Subsets of components of the EPS plant and number of variables associated with the related contracts, including communications variables and variables related to the health status of plant components (e.g. buses, contactors).

Each contract in our library specifies a "local" controller for a portion of the EPS plant, i.e., a subset of its components, derived from the specifications in Section 4.3. In addition to sensing (input) and actuation (output) variables, here contracts include a set of communication variables to propagate information on error conditions and component health status. Figure 5.4 shows some of the EPS subsystems supported by our library. A contract for the subsystem in Figure 5.4.a) specifies that the contactor should be opened and the failure variable asserted if the generator fails; otherwise the contactor must be closed. For the subsystem in Figure 5.4.c), the same requirement as for Figure 5.4.a) will hold, with the addition that both generators should never be connected at the same time to avoid paralleling AC sources. For the subsystem in Figure 5.4.e), we require that the contactor on one side should be closed upon reception of a failure signal from a component connected to the opposite side. The contract for the subset in Figure 5.4.b) specifies that the load should be isolated in case of failures in one of the interconnected portions of the plant, or in the rectifier unit. Finally, the subsystem in Figure 5.4.d) is associated to a contract similar to the one in Figure 5.4.e), while handling one additional bus and only two interconnection branches. For each portion of the plant, the library can provide multiple contracts to specify different sets of behaviors. Moreover, we provide contracts that specify abstractions of controllers for specific portions of the plant. For example, a contract may represent the behavior of the controller associated to an idealized generator, such as the component A_1 in Table 4.2, which abstracts both the sub-systems in Figure 5.4.a) and 5.4.c). Overall, the library includes 17 contracts and 9 refinement assertions. The verification of the refinement assertion using NuSMV required 1.55 s.

A controller for the EPS has been assembled out of 5 different contracts from the library, associated to the subsystems shown in Figure 5.4. The composite contract has a total of 46 variables. On the other hand, the most compact abstraction of the design based on the available library had only 12 variables. On this design, we checked the following properties expressed as LTL A/G contracts (cf. Table 4.1 for details):

- $C_{p1}, ..., C_{p4}$: If generator $G \in \{G_L, A_L, A_R, G_R\}$ fails, the closest contactor $c \in \{c_1, ..., c_4\}$ must be opened;
- C_{p5}: If generators G_L and G_R are healthy, contactors c_5 and c_6 must be opened;
- C_{p6}: Contactors c_2 and c_3 cannot be both closed at the same time;
- $C_{p7}, ..., C_{p10}$: If at least one generator is healthy, AC loads cannot be unpowered (variations of specification S_7 of Table 4.1);
- C_{p11}: If all generators are healthy, bus B_5 must not be powered (variation of specification S_6 of Table 4.1);
- C_{p12}, C_{p13}: If at least one generator is healthy, c_{11} and c_{12} cannot stay opened for more than three clock cycles.

This set of property contracts has been verified using both the RCPL and the RCP algorithms. The total execution time was 123.1 s for RCPL, and 638.82 s for RCP. Figure 5.5 shows

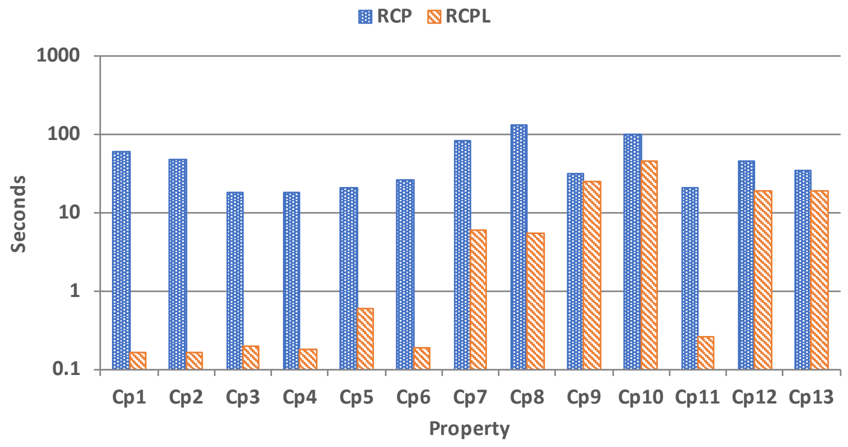


Figure 5.5: Execution time of RCPL and RCP algorithms for the EPS case study for the verification of the set of 13 property contracts

the execution times required by each verification task. For more than half of the properties ($C_{p1}, C_{p2}, C_{p3}, C_{p4}, C_{p5}, C_{p6}, C_{p11}$), RCPL allows to obtain a performance improvement of two orders of magnitude, by using an abstraction of the controller with only 12 variables. For C_{p7} and C_{p8} RCPL shows a performance improvement of one order of magnitude, while for the other properties, the execution times are comparable to the one obtained with plain RCP. C_{p10} produced the worst execution time, using an abstraction with 37 variables. Figure 5.6

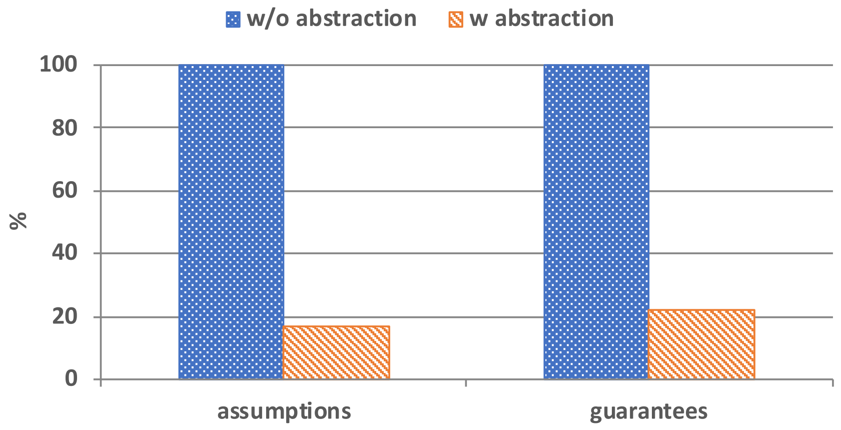


Figure 5.6: LTL formula size ratio of the abstract EPS contract w.r.t. the non-abstract version

shows the difference in terms of formula sizes, computed as the ratio between the non-abstract

EPS contract size and the one of its maximal abstraction obtained at the first iteration of the `BuildAbstraction` algorithm. Formulas in abstract contracts are indeed smaller than the original ones, which provides an explanation of the performance improvement obtained using RCPL.

To test the scalability of the algorithm, the same properties have been checked on an extended plant architecture, including one more generator, 7 contactors, 2 rectifier units, 2 AC loads, 2 DC loads and one bus. The contract specifying a controller for the new plant includes 66 variables. Verification of the whole property set was performed in 1724.43 s with RCPL and 8371.01 s with RCP. Also in this example, an execution time two orders of magnitude smaller for RCPL has been observed. In the best case, the generated abstract contract included only 16 variables.

5.5 Conclusion

We addressed the problem of performing scalable refinement checks for contract-based design. We presented an algorithm that leverages a pre-characterized library of contracts enriched with refinement assertions to break the main verification task into a set of smaller tasks. We applied the proposed algorithm to verify controllers for aircraft electrical power systems, with up to two orders of magnitude improvement with respect to a standard implementation based only on LTL satisfiability solving. A full-fledged theoretical study of its complexity is challenging, since its runtime is highly dependent on the characteristics of the library, in addition to the structure of the system and the property under consideration. A characterization of the role of the library via domain-related benchmarks will be object of future work. We here anticipate that the benefits of having a richer library in terms of refinement assertions will largely repay the overhead of building it. In fact, we recall that the library verification process must be performed only once, outside of the main verification flow. Moreover, the proposed algorithm already offers a way of automatically proving new refinement relations that can be effectively used to further populate the original library so as to enrich it for future verification tasks.

Chapter 6

Constrained Synthesis from Libraries of Generic Components

In general, synthesis from libraries of components is the problem of building a network of components from a given library, such that the network realizes a given specification. This is an undecidable problem. It becomes decidable if we impose a bound on the number of chosen components. However, the bounded problem remains computationally hard and brute-force approaches do not scale.

In this chapter, we study methods for solving the problem of bounded synthesis from libraries of generic components. By saying generic, we mean that we do not assume any specific formalism describing the components, only requiring to be able to connect them together (composition) and comparing them (refinement). Our solution is based on the Oracle-Guided Inductive Synthesis (OGIS) paradigm. Although our synthesis algorithm does not assume a specific formalism *a priori*, we present a parallel implementation which, concretely, instantiates components defined as LTL A/G contracts. We show the potential of our approach and evaluate our implementation by applying it to two industrial-relevant case studies.

The rest of the chapter is organized as follows. We introduce the problem and discuss the motivation for this work in Section 6.1, while in Section 6.2 we define the synthesis problem we tackle and analyze its complexity, introducing a running example to explain in detail the problem encoding and the approach we adopt. We propose a solution for a concrete version of the problem in Section 6.3 and discuss implementation aspects in Section 6.4, including the description of the parallel variant of our algorithm. In Section 6.5 we present the case studies and empirical results. and we draw conclusions in Section 6.6.

6.1 Introduction

Synthesis from component libraries is the process of synthesizing a new component by composing elements chosen from a library. This type of synthesis is able to capture the complexity of CPS by restricting possible synthesis outcomes to a set of well-tested, already available components.

However, the general problem of synthesis from component libraries, where the components are state machines, is undecidable [54]. In this chapter, we focus on a decidable variant of the problem, where an explicit bound on the number of components in a solution is provided. Our goal is to find a composition of components which satisfies a specification, when no assumption on the formalism used to describe those components is made, besides requiring the ability to connect components to allow them working together (composition), and comparing them (refinement).

Although no particular formalism is assumed *a priori*, we cast a concrete version of this problem using LTL A/G contracts as the underlying specification of components. We then show how it is possible to solve this problem by presenting two variants of an algorithm, a sequential and a parallel one, based on the *Oracle-Guided Inductive Synthesis* (OGIS) paradigm [49]. To reduce the solution search space, this algorithm leverages designer hints, types, and other constraints over components, possibly *precomputed* and stored in the libraries as additional composition rules. To the best of our knowledge, this is the first time that a concurrent synthesis algorithm is proposed for this problem, thanks to the decoupling of a solution *topology* from its semantic evaluation.

Developing a synthesis algorithm which is agnostic of the formalism of the component specifications is a crucial characteristic of the methods discussed in this chapter. Often, in fact, OGIS approaches rely on sets of input values to describe counterexamples. This allows to effectively guide the synthesis process, but it sacrifices flexibility as the input format needs to be fixed, and it requires an oracle able to return a valid input as a counterexample. Additionally, it is a problem in cases in which the input space is infinite, or when each input has infinite size, as in the case of LTL. Here, we focus on the definition of a general synthesis strategy that relies on an oracle which is only required to answer *yes* or *no* to a candidate solution. In this case, the counterexample we look at is the candidate composition itself, when it has been declared incorrect by the oracle. Later, in Chapters 7 and 8, we will relax this constraint and present techniques that take full advantage of component formalisms, albeit requiring a more capable oracle.

The implementation of the algorithms discussed in this chapter resulted in a tool called PYCO, able to exploit multiprocessor computer architectures to speed up synthesis. We evaluate PYCO by synthesizing two industrial-relevant designs: first, the controller of a *Brushless DC Electric Motor* (BLDC), including both architectural and software aspects, and then, the controller for an aircraft *Electrical Power distribution System* (EPS), introduced in Chapter 4. This last problem, in particular, has already been studied using contracts [41, 61]. In these papers, however, contracts have been used mostly for verification and to describe requirements, without playing any role in the controller synthesis process itself, performed using standard reactive synthesis techniques. Here, contracts collected in the component library represent controllers for a number of EPS subsystems. Our synthesis algorithm, for the first time, operates directly on those contracts to compose a controller that satisfies all the requirements.

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

6.2 Constrained Synthesis from Component Libraries (CSCL)

In our framework, a *component* $G \in \mathbb{G}$, where $\mathbb{G}$ is the domain representing the space of all possible components, is a tuple $G = (I_G, O_G, \delta_G, \sigma_G, R_G)$. I_G is the set of input ports, O_G is the set of output ports, and δ_G is the component specification, expressed using a specific notation (e.g., an A/G contract, or an LTL formula). Variables in δ_G correspond to ports in I_G and O_G. I_G, O_G, and δ_G are all defined over a common set of symbols, or alphabet, Σ_{IO}. The function $\sigma_G : I_G \cup O_G \to T$ maps ports of G to elements in T, where T is a *typeset*. A typeset is a poset consisting of a set of symbols (types) ordered by the *subtype* relation[1]. For $a, b \in T$, the notation $a \leq b$ means that b is a subtype of a. Finally, R_G is a set of logic constraints over ports of G. We require that any two component specifications, say δ_1 and δ_2, can be composed as $\delta = \delta_1 \parallel \delta_2$. We also indicate refinement of δ_2 by δ_1 as $\delta_1 \sqsubseteq \delta_2$[2]. From Chapter 3, however, we borrow the renaming mechanisms. Thus $\delta' = \delta\theta$ indicates the component specification δ where the variables have been renamed according to θ.

A component library, reminiscent of a contract library defined in Section 3.3, is a tuple $L = (Z, T, R_Z, f)$. Here, $Z = \{G_1, \ldots, G_n\}$ is a finite set of components. Components in Z are required to have unique ports (and variables), meaning that they are not shared with other components. R_Z is a set of logic constraints that encode connection rules over ports of components in Z and types in T. Constraints in R_G and R_Z characterize a certain library, and are used by the library designer to provide domain-specific insights that can be used to speed up the synthesis process. The cost function $f : \wp(\mathbb{G}) \to \mathbb{R}$, where $\wp(\mathbb{G})$ is the powerset of $\mathbb{G}$, maps sets of components from the domain $\mathbb{G}$ to real numbers, i.e., the cost of the component.

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

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

[2]We use the symbols $\parallel, \sqsubseteq$ to describe the composition and refinement of generic components, distinguishing them from the contract composition and refinement, indicated with the symbols $\otimes, \preceq$.

Component connections are defined not only by constraints in R_Z and R_Gs, but they also need to be sound according to some general composition rules. For instance, we will never allow two components to control the same output ports. These general composition rules between components, described later in Section 6.2.2, are collected in a set called Q. These rules are constraints that need to be applied no matter what a specific library defines.

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

To simplify our discussion, for a library $L = (Z, T, R_Z, f)$, we will use the following set as a shortcut to address all the ports of all components in it:

$$\mathcal{P}_{lib} = \bigcup_{i=1}^{|Z|} I_{G1} \cup O_{G1} \tag{6.1}$$

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

$$\mathcal{P}_{lib \cup S} = \mathcal{P}_{lib} \cup I_S \cup O_S \tag{6.2}$$

The composition of two components $G_1 = (I_1, O_1, \delta_1, \sigma_1, R_1)$ and $G_2 = (I_2, O_2, \delta_2, \sigma_2, R_2)$ is a new component $G_1 \parallel G_2 = ((I_1 \cup I_2) \backslash (O_1 \cup O_2), O_1 \cup O_2, \delta_1 \parallel \delta_2, \sigma_1 \cup \sigma_2, R_1 \cup R_2)$, assuming that composition is also defined for δ_1 and δ_2. This means that ports that are shared between input and output sets of the components, i.e., they are connected, are considered outputs in the resulting composition. For instance, when input a is connected to output b, the resulting composite component only contains output b (input a "disappears" since it is going to be controlled by b). We also assume that there is no conflict between σ_1 and σ_2, meaning that ports with the same name need to have the same type according to both σ_1 and σ_2:

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

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

$$I_1 \subseteq I_2, O_2 \subseteq O_1, \text{ and } \delta_1 \sqsubseteq \delta_2 \tag{6.3}$$

where we assume that the formalism used to express component specifications δ_1 and δ_2 includes the notion of refinement. For instance, if δ_1 and δ_2 are logic formulas, $\delta_1 \sqsubseteq \delta_2$ is equivalent to the implication $\delta_1 \to \delta_2$. Intuitively, if G_1 refines G_2, then δ_2 will always hold if δ_1 holds, i.e., G_1 can be safely used in place of G_2.

As in the case for connection of component specifications, we lift the definitions of interconnection through θ to apply to generic components, too. That is, renaming a component $G = (I_G, O_G, \delta_G, \sigma_G, R_G)$ according to a certain θ will yield $G\theta = (I_{G_\theta}, O_{G_\theta}, \delta_{G\theta} = \rho_\theta \Rightarrow \delta_G,$

$\sigma_{G_\theta}, R_{G_\theta})$, where

$$\begin{aligned}
I_\theta &= (I \cup \left\{\ y \quad \text{if } \exists y\colon (x,y) \in \theta \ \middle|\ x \in I\right\}) \setminus O_\theta \\
O_\theta &= O \cup \left\{\ x \quad \text{if } \exists y\colon (x,y) \in \theta \ \middle|\ x \in I\right\} \cup \left\{\ y \quad \text{if } \exists y\colon (x,y) \in \theta \ \middle|\ x \in O\right\} \\
\rho_\theta &= \bigwedge_{x,y \in I_\theta \cup O_\theta, (x,y) \in \theta} x \equiv y
\end{aligned}$$

and where $\delta_{G\theta} = \rho_\theta \Rightarrow \delta_G$ is the component specification reflecting the renaming implied by ρ_θ within the context of the concrete formalism of δ_G, and $\sigma_{G_\theta}, R_{G_\theta}$ are updated accordingly. For instance, by renaming a component $G = (\emptyset, \{a, b\}, \delta_G, \emptyset, \emptyset)$, where δ_G is a LTL formula, according to $\theta = \{(a, b)\}$ we get $G\theta = (\emptyset, \{a, b\}, \Box(a = b) \to \delta_G, \emptyset, \emptyset)$. In Chapter 3, instead, we describe the details of the interconnection when the component specifications are A/G contracts. Given two components G_1, G_2, we say that G_1 is equivalent to G_2 if and only if there exists an interconnect, or renaming, θ', such that $G_1\theta' \sqsubseteq G_2\theta'$ and $G_2\theta \sqsubseteq G_1\theta$.

Hence, we can express logic constraints in R_Z, R_G, and R_S, and model interactions between components (e.g., when the output of a component is the input of another one) in terms of properties of an interconnect θ, which will be used to define connections among components in the library. We will write $\theta_{x,y}$ to indicate that a certain connection is logically implied by the interconnect, i.e., $\rho_\theta \Rightarrow x \equiv y$. On the other hand, we might want to indicate that a certain renaming is not implied by θ. Thus, we write $\neg\theta_{x,y}$ to indicate that $\rho_\theta \not\Rightarrow x \equiv y$.

Note that using θ to define connections between components can, potentially, yield inconsistent renaming of variables and thus inconsistent compositions of components. Consider, for instance, three ports p, q, t and a function θ such that $\theta_{q,p}$, $\theta_{t,p}$, and $\neg\theta_{q,t}$. Clearly, no such renaming could be applied because the first two connections imply $q \equiv t$. In Section 6.2.2, we describe how to properly constrain θ to avoid such situations.

Example 4 (Component Connection). *Let*

$$\begin{aligned}
G_1 &= (I_1, O_1, \delta_1, \sigma_1, R_1) = (\{a_1, b_1\}, \{c_1\}, c_1 = a_1 + b_1, \{a_1, b_1, c_1\} \to \{\bot\}, \emptyset) \\
G_2 &= (I_2, O_2, \delta_2, \sigma_2, R_2) = (\{a_2\}, \{b_2\}, b_2 = 2 \cdot a_2, \{a_2, b_2\} \to \{\bot\}, \emptyset)
\end{aligned}$$

be two components and θ a renaming function specifying a single connection θ_{b_1,b_2} (thus we have also $\neg\theta_{a_1,b_2}, \neg\theta_{c_1,b_2}, \neg\theta_{a_1,a_2}$, etc.). Let us also assume that component specifications can be composed by taking their conjunction. Then, the composition $(G_1 \parallel G_2)\theta$ yields a component

$$\begin{aligned}
(G_1 \parallel G_2)\theta =& (\{a_1, a_2\}, \{c_1, b_1, b_2\}, \\
& b_1 = b_2 \wedge b_2 = 2 \cdot a_2 \wedge c_1 = a_1 + b_1, \{a_1, a_2, b_2, b_1, c_1\} \to \{\bot\}, \emptyset)
\end{aligned}$$

Example 5 (Running example: synthesize the modulo operation). *We introduce here a simple example to help the reader familiarize with the concepts introduced so far. Our objective is to synthesize the modulo operation starting from a library of simpler arithmetic operations. For simplicity, we assume only strictly positive integer inputs.*

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

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

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

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

The specification is the component $S_{mod} = (\{x, y\}, \{z\}, z = mod(x, y), \{x, y, z\} \rightarrow \{\perp\}, \emptyset)$. We know that the modulo operation can be computed as $mod(x, y) = x - \lfloor x/y \rfloor \cdot y$. A composition of elements in Z_{op} that implements S_{mod} is shown in Figure 6.1: $sub(x, mult(div(x, y), y))$, with connections θ_{b_s,c_m}, θ_{a_m,c_d}, θ_{x,a_s}, θ_{x,a_d}, θ_{y,b_d}, θ_{y,b_m}, θ_{z,c_s}.

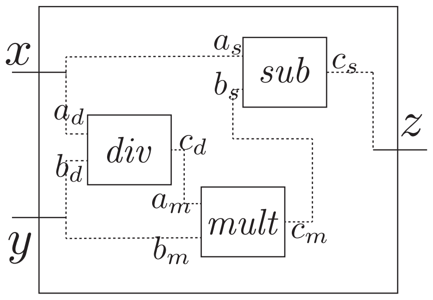


Figure 6.1: Modulo operation composition from elements in L_{op}.

In the following sections, we will define a set of rules to automatically obtain candidate solutions which are topologically sound (e.g., adding the connection θ_{z,c_m} should be illegal because the output z is already connected, or controlled, by the port c_s), and semantically correct (e.g., having θ_{z,c_m} instead of *θ_{z,c_s} would yield a composition which does not implement the modulo operation, although topologically sound).*

6.2.1 A combinatorial analysis of CSCL

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

First, we examine the case in which we already have a set of m components $H = \{G'_1, \dots, G'_m\}$, where no additional variable renamings need to be specified, and we want to find a single component $G_z \in Z$ such that $\delta_z \parallel \delta'_1 \parallel \cdots \parallel \delta'_m \sqsubseteq \delta_S$. Assuming n is the number of components in Z, we have n possibilities to try. Extending this example to include $c \leq n$ unknown components is straightforward. In this case, there are $\frac{n!}{c!(n-c)!}$ possible solutions, assuming that the order of components does not matter.

On the other hand, we have a scenario in which we still have m components $H = \{G'_1, \dots, G'_m\}$, but connections among them are missing. We want to connect the components to each other, according to a certain function θ, such that $(\delta'_1 \parallel \cdots \parallel \delta'_m)\theta \sqsubseteq \delta_S$. The complexity of this problem depends on the total number of ports. Assuming p is the number of ports of a component, then there are $2^{\frac{mp(mp-1)}{2}}$ possible solutions.[3]

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

6.2.2 Synthesis Constraints

The analysis in Section 6.2.1 shows that the CSCL problem grows quickly with the number of components and ports in the library. The role of the library-specific constraints is to mitigate such complexity. We require R_Z to contain *at least* the constraints defined in the following paragraphs[4]:

- Connections must be consistent, according to the following properties which encode the semantics of θ. Equation 6.4 tells us that if for three ports p, q, w we have $\theta_{p,q}$ and $\theta_{q,w}$, then it must be also $\theta_{p,w}$:

$$\forall p, q, w \in \mathcal{P}_{lib \cup S} : \theta_{p,q} \wedge \theta_{q,w} \Rightarrow \theta_{p,w} \tag{6.4}$$

  Equation 6.5 represents the fact that, logically, if p is connected to q, then q is also connected to p:

$$\forall p, q \in \mathcal{P}_{lib \cup S} : \theta_{p,q} \Rightarrow \theta_{q,p} \tag{6.5}$$

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

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

Equation 6.6 simply states that a port is always connected to itself:

$$\forall p \in \mathcal{P}_{lib \cup S} : \theta_{p,p} \tag{6.6}$$

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

$$\forall G, G' \in Z : \forall p, q \in O_G \cup O_{G'} : (p \neq q) \Rightarrow \neg\theta_{p,q} \tag{6.7}$$

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

$$\forall G \in H : \forall p \in I_G : (\exists s \in I_S : \theta_{p,s}) \vee (\exists G' \in H : \exists q \in O_{G'} : \theta_{p,q}) \tag{6.8}$$

Example 6. *Equation 6.7 prevents the connection between multiple outputs of components in H. With respect to Ex. 5, this means enforcing $\neg\theta_{c_a,c_s}$, $\neg\theta_{c_a,c_m}$, $\neg\theta_{c_a,c_d}$, $\neg\theta_{c_s,c_m}$, $\neg\theta_{c_s,c_d}$, and $\neg\theta_{c_m,c_d}$. Equation 6.8, instead, makes sure that no inputs of the components in H are left unconnected. For instance, the composition in Figure 6.2 violates Equation 6.8, because a_s is not connected to any other port.*

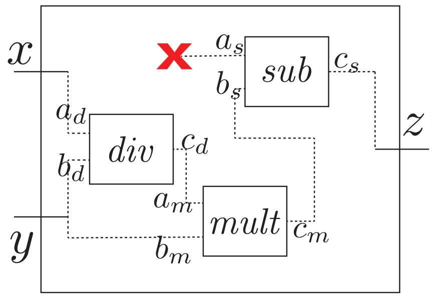


Figure 6.2: Illegal composition of elements in L_{op} (a_s disconnected).

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

$$\forall s, r \in I_S \cup O_S : s \neq r \Rightarrow \neg\theta_{s,r} \tag{6.9}$$

- Inputs of the specification S cannot be connected to component outputs, because otherwise in the resulting composition those inputs will be treated as outputs (as seen in Section 6.2):

$$\forall s \in I_S : \forall G \in Z : \forall p \in O_G : \neg\theta_{s,p} \tag{6.10}$$

- Every input of the specification S has to be connected at least to an input of a component in H (Equation 6.11), while every output of S has to be connected at least to an output of a component in H (Equation 6.12):

$$\forall s \in I_S : \exists G \in H : \exists p \in I_G : \theta_{s,p} \tag{6.11}$$

$$\forall s \in O_S : \exists G \in H : \exists p \in O_G : \theta_{s,p} \tag{6.12}$$

Example 7. *Equation 6.11 and Equation 6.12 ensure that there is a full mapping of specification ports into components ports. For instance, the composition in Figure 6.3 violates Equation 6.12 because there is an output of the specification, z, which is not connected to any component outputs.*

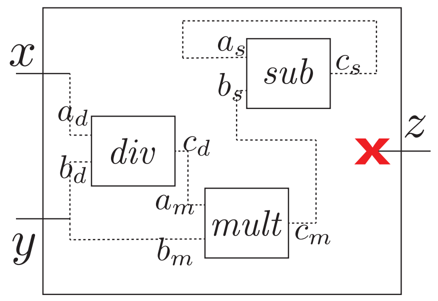


Figure 6.3: Illegal composition of elements in L_{op} (z disconnected).

- Only ports with compatible types can be connected to each other, according to the subtype relation defined in Section 6.2 and considering contravariant inputs and outputs. This means that, given two ports p and q connected to each other, if p is an output and q is an input, then $\sigma(p) \leq \sigma(q)$, and *vice versa* (similarly, in principle, to what is described by de Alfaro and Henzinger in [1]):

$$\forall G, G' \in Z : \forall p \in I_G, q \in O_{G'} : \sigma_G(p) \not\leq \sigma_{G'}(p) \Rightarrow \neg\theta_{p,q} \tag{6.13}$$

6.2.2.1 Problem Definition

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

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

2. requiring the library of components to satisfy *at least* the constraints presented in Equations 6.4 to 6.13, allowing the designer, however, to add as many constraints as necessary, according to the problem domain.

Definition 8 (CSCL problem). *Let $S = (I_S, O_S, \delta_S, \sigma_S, R_S)$ be a system specification, and $L = (Z, T, R_Z, f)$ a library of components where R_Z contains at least the constraints described in Equations 6.4 to 6.13. Let also the operations of composition ($\|$) and refinement ($\sqsubseteq$) be defined for components in L. The problem of Constrained Synthesis from Component Libraries consists of finding a finite set of components $H = \{G_1, \ldots, G_N \mid G_i = (I_i, O_i, \delta_i, \sigma_i, R_{Gi}) \in Z\}$ and a interconnect θ such that the cost function f is minimized according to:*

$$\underset{\{G_1,\ldots,G_N\}}{\text{minimize}} \quad f(\{G_1, \ldots, G_N\}) \tag{6.14a}$$

$$\text{subject to} \quad \textit{all synthesis constraints in } R_Z,\ R_S,\ R_{Gi}\ , \tag{6.14b}$$

$$(\delta_1 \parallel \cdots \parallel \delta_N)\theta \sqsubseteq \delta_S \tag{6.14c}$$

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

Definition 9 (Simplified CSCL problem). *Let $S = (I_S, O_S, \delta_S, \sigma_S, R_S)$ be a system specification, and $L = (Z, T, R_Z, f)$ a library of components where R_Z contains at least the constraints described in Equations 6.4 to 6.13, and f is a constant. Let also the operations of composition ($\|$) and refinement ($\sqsubseteq$) be defined for components in L. The simplified CSCL problem consists of finding a finite set of components $H = \{G_1, \ldots, G_N \mid G_i = (I_i, O_i, \delta_i, \sigma_i, R_{Gi}) \in Z\}$ and a interconnect θ such that:*

$$\textit{all synthesis constraints in } R_Z,\ R_S,\ R_{Gi}\ \textit{hold} \tag{6.15a}$$

$$(\delta_1 \parallel \cdots \parallel \delta_N)\theta \sqsubseteq \delta_S \tag{6.15b}$$

6.3 Solving a concrete instance of the CSCL problem

The CSCL problem in Def. 8 (and 9) is very general, and the most effective approach to solving it depends on the structure of the library. For instance, a continuous cost function f will require optimization techniques which are very different from a cost function which is purely discrete, or which depends on the formalism used to describe component specifications.

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

- f can be solved using discrete optimization techniques;
- f does not depend on the formalism used to describe the specification of components. This means that, for a component G, $f(G)$ can be evaluated without considering δ_G.

This choice allows us to effectively decouple the topological aspects of a candidate solution of the CSCL problem in Def. 8, i.e., Equation 6.14a and 6.14b, from its semantic evaluation, i.e., Equation 6.14c.

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

STEP 1 A discrete optimization problem is solved to retrieve a candidate solution, that is, a set of components, and their connections, which minimizes the objective function and satisfies all the synthesis constraints. With respect to Def. 8, this first step takes care of Equation 6.14a and 6.14b, and provides the function θ used to solve Equation 6.14c. This step, in general, can be solved by a constraint solver. With respect to Example 5, for instance, this step corresponds to the generation of possible solution candidates such as $\mathit{sub}(x, \mathit{mult}(\mathit{div}(x, y), y))$, or $\mathit{add}(x, \mathit{mult}(\mathit{div}(x, y), y))$.

STEP 2 Equation 6.14c is checked by interrogating a tool, which we call *verifier*, able to *understand* component specifications. The verifier determines whether the candidate composition, after proper interconnection of components, refines the global specification δ_S. In Example 5, the verifier is the tool able to determine, indeed, that $\mathit{sub}(x, \mathit{mult}(\mathit{div}(x, y), y))$ implements the modulo operation.

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

We call *counterexample* a candidate composition (i.e., a set of components and their connections) which has been proven wrong by the verifier. A counterexample is used to *inductively* learn new constraints for the solver. In general, the performance of an OGIS-based algorithm depends on how well the information provided by the oracle helps prune the search space, leveraging as much as possible the information that can be inferred from the components specifications and execution traces, if the verifier provides them. In this work, however, we only assume the verifier is able to check the validity of candidates, returning a yes/no answer. Our solution leverages component equivalence, introduced in Section 6.2. We can do so by using the refinement operation that we assume in Definitions 8 and 9, thus preserving the generality of the approach.

We can indicate equivalent components using the function

$$E : \mathbb{G} \to 2^{\mathbb{G} \| \Theta} \tag{6.16}$$

which takes a component as input and returns a set of pairs, consisting of a component and an interconnect. Given a component G in a library L, $E(G)$ returns a set containing all the pairs (G', θ') such that G is equivalent G' according to θ'. Formally,

$$E(G) = \left\{ (G', \theta') \;\middle|\; \begin{array}{ll} G' \in Z & \text{and} \\ G\theta' \sqsubseteq G'\theta' & \text{and} \\ G'\theta' \sqsubseteq G\theta' & \end{array} \right\} \tag{6.17}$$

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

6.3.1 The `CSCL` algorithm

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

A number of constraint solvers are able, indeed, to minimize objective functions while satisfying logic constraints, such as Z3 [60]. The simpler formulation of the CSCL problems in Def. 9, on the other hand, doesn't require the function f to be minimized and can be solved by using a constraint solver without optimization capabilities.

In our synthesis strategy, illustrated in Algorithm 5, the task of pruning the search space is carried out in a twofold manner. First, the constraint solver only needs to search over a number of potential candidates which is drastically limited by the synthesis constraints. Such constraints include those encoded in the library through the constraints in R_Z, $R_{G_1}, \ldots, R_{G_N}$, and R_S.

Second, each time a counterexample is observed (see the block starting at Line 9) it is used to match all the elements of the library equivalent to those in the counterexample, according to the component equivalent sets described in Equation 6.17, and Algorithm 6. This allows us to rule out a number of possible candidate instances exponential in the number of components contained in the rejected candidate. For instance, if the counterexample is a composition of 4 components, and for each one there are 2 other components in Z with the same specification, then adding constraints from that single counterexample will discard 3^4 erroneous candidate instances.

The output of the `CSCL` algorithm is a finite set of components, H, and their connections, expressed as an interconnect θ. To ensure termination, the algorithm requires a bound on the number of components used in a candidate solution. The choice of such bound depends on the details of the problem being solved. Here, we let the user decide what is the most appropriate bound through the input parameter N.

The complexity of the `CSCL` algorithm depends on the structure of the library and the solution maximum size. According to our analysis in Section 6.2.1, for a fixed library, the worst case time complexity will be exponential in the maximum number of components in

```
1  function CSCL:
       Input: specification $S = (I_S, O_S, \delta_S, \sigma_S, R_S)$, library of components
              $L = (Z, T, R_Z, f)$, maximum number of components in the solution $N$
       Output: set of components $H = \{G_1, \ldots, G_n\}, H \subseteq Z$ and $n \leq N$, connection
               function $\theta$ such that $(\delta_1 \parallel \cdots \parallel \delta_n)\theta \sqsubseteq \delta_S$ and $f$ is minimized, or $False$ if no
               solution is found
2      Initialize constraint solver and verifier, instantiating the synthesis constraints for problem
        instance and setting $N$ as the maximum number of components in a candidate
        solution.;
3      (STEP1) while get candidate solution $(H' = \{G_1, \ldots, G_n\}, \theta')$, with $n \leq N$, from
        constraint solver such that $f$ is minimized and all the synthesis constraint hold do
4          Build composition $(G_1 \parallel \cdots \parallel G_n)\theta'$;
5          (STEP2) if verifier checks that $(\delta_1 \parallel \cdots \parallel \delta_n)\theta' \sqsubseteq \delta_S$ holds then
6              $H \leftarrow H'$;
7              $\theta \leftarrow \theta'$;
8              return $(H, \theta)$;
9          else
10             $R \leftarrow$ RejectCandidate($H', \theta'$);   // infer equivalent erroneous
                candidates, see Algorithm 6
11             foreach $(H_t, \theta_t) \in R$ do
12                 add constraint $(H, \theta) \neq (H_t, \theta_t)$ to constraint solver;
13             end
14         end
15     end
16     return $False$;
17 end
```

Algorithm 5: CSCL algorithm. (STEP 1) and (STEP 2) are labels. The algorithm interfaces with a constraint solver (preferably with optimization capabilities) to execute (STEP 1), and with a verifier, e.g., a model checker, to execute (STEP 2).

```
function RejectCandidate:
    Input: set of contracts $H = \{G_1, \ldots, G_n\}$, interconnect $\theta$
    Output: set containing interconnects $R$

    $R \leftarrow \{\}$;
    forall $(G'_1, \theta'_1) \in E(G_1), (G'_2, \theta'_2) \in E(G_2), \cdots, (G'_n, \theta'_n) \in E(G_n)$ do
        $H' \leftarrow \{G'_1, \cdots, G'_n\}$;
        initialize new interconnect $\theta'$;
        foreach $(p, q) \in \theta$ do
            $G_i \leftarrow$ component in $H$ such that $p \in I_i \cup O_i$;
            $G_j \leftarrow$ component in $H$ such that $q \in I_j \cup O_j$;
            add $(\bar{\theta}'_i(p), \bar{\theta}'_j(q)))$ to $\theta'$;        // for $\bar{\theta}$ see Section 3.2.1.6
        end
        add $(H', \theta')$ to $R$;
    end
    return $R$;
end
```

Algorithm 6: RejectCandidate algorithm. The returned set R collects all the possible candidates equivalent to (H, θ).

a solution, N, multiplied by the complexity of the call to the verifier. In practice, though, the techniques discussed above are able to limit the search space in a considerable manner, yielding acceptable synthesis times in many cases.

6.3.2 An efficient representation for the synthesis constraints

The encoding of the synthesis constraints presented in Section 6.2.2 is not particularly efficient. For instance, one can see how Equation 6.4 represents a formula which grows cubically in the number of ports of all components in the library. Such encoding would cause the internal representation of the synthesis constraints in the constraint solver to grow unnecessarily large, resulting in poor performance. In this section, we present an encoding that exploits a more efficient representation of component connections.

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

$$\begin{aligned} &\forall G \in Z : \forall p \in I_G : \\ &\quad (\mathcal{M}(p) = -1) \vee [\exists G' \in Z : \exists q \in O_{G'} : \mathcal{M}(p) = \mathcal{I}(q)] \vee [\exists s \in I_S : \mathcal{M}(p) = \mathcal{I}(s)] \end{aligned} \tag{6.18}$$

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

$$\forall s \in O_S : (\mathcal{M}(s) = -1) \vee (\exists G \in Z : \exists p \in O_G : \mathcal{M}(s) = \mathcal{I}(p)) \tag{6.19}$$

The following theorem states that using the encoding presented in this section, with Equations 6.19 and 6.18, yields a solution space which is at least as large as the one obtained representing θ using Equations 6.4 to 6.13. Our goal is to show that if a solution exists within the constraint in Equations 6.4 to 6.13, then it will exist also using the encoding introduced here.

Theorem 6.3.1. *Let* $\mathbf{C_1}$ *be the set of connections among components in* $Z \cup \{S\}$ *that can be defined by the function* θ *according to Equations 6.4 to 6.13. Let also* $\mathbf{C_2}$ *be the set of connections that can be defined by the connection variables in* $\mathbb{V}$ *and indices in* $\mathbb{I}$*, constrained by Equations 6.18 and 6.19. Then* $\mathbf{C_1} \subseteq \mathbf{C_2}$*.*

Proof. We start considering only connections between ports in $\mathcal{P}_{lib}$. Given an input port p and an output port q, a connection $\theta_{p,q}$ in $\mathbf{C_1}$ (and by Equation 6.5 also $\theta_{q,p}$) can be trivially be represented in $\mathbf{C_2}$ by the assignment $\mathcal{M}(p) = \mathcal{I}(q)$. If both p and q are outputs, then by Equation 6.7 their connection cannot be in $\mathbf{C_1}$. If both p and q are inputs and $\theta_{p,q}$ is in $\mathbf{C_1}$, then by Equation 6.8 p and q have to be connected to another output in the library or to an input of S. In either case, assume w be such port, where $\theta_{p,w}$ and $\theta_{q,w}$ are also in $\mathbf{C_1}$. Then $\mathcal{M}(p) = \mathcal{I}(w)$ and $\mathcal{M}(q) = \mathcal{I}(w)$ represent the equivalent connections in $\mathbf{C_2}$, including indirectly $\theta_{p,q}$ (because they have a reference to the same index). Consider now also ports of the specification S. Since, in $\mathbf{C_1}$, we do not allow any two ports of S being connected to each other (Equation 6.9), we have only the case in which there is a connection $\theta_{s,p}$ between ports $s \in I_S \cup O_S$ and $p \in \mathcal{P}_{lib}$. If s is an input, then p has to be an input too (because of Equation 6.10), and we can represent $\theta_{s,p}$ as $\mathcal{M}(p) = \mathcal{I}(s)$ in $\mathbf{C_2}$. If s is an output, then p can be either a component input or output. If p is an output, then $\theta_{s,p}$ can be represented as $\mathcal{M}(s) = \mathcal{I}(p)$. If p is an input, then by Equation 6.12 there must be another component output q such that $\theta_{s,q}$. By Equation 6.4, then it must be also $\theta_{p,q}$. Therefore we can map these three connections in $\mathbf{C_2}$ with $\mathcal{M}(s) = \mathcal{I}(q)$ and $\mathcal{M}(p) = \mathcal{I}(q)$ (where $\theta_{s,q}$ is implicit because s and p refer to the same index). This shows that all the connections in $\mathbf{C_1}$ have an equivalent in $\mathbf{C_2}$, hence $\mathbf{C_1} \subseteq \mathbf{C_2}$. $\square$

Thanks to Theorem 6.3.1, we are ensured that the encoding presented here (under Equations 6.18 and 6.19) preserves the solution space defined by Equations 6.4 to 6.13. All the results in Section 6.5 are obtained after reformulating the synthesis constraints in Equations 6.4 to 6.13 using the encoding described in this section.

6.4 Implementing the CSCL algorithm

In this and the following sections, we describe the implementation of a parallel variant of the CSCL algorithm and evaluate its capabilities and performance. We used the Satisfiability Modulo Theories (SMT) solver Z3[60, 9] to find candidates satisfying the synthesis constraints and minimize the cost function f, and we chose to represent our library as a set of LTL A/G contracts. This choice is also motivated by the fact that composition and refinement operations are well defined in the contract algebra. Moreover, additional concepts such as compatibility and consistency can be leveraged to derive, before the actual synthesis process, library constraints on components composability (in the form of incompatible sets of ports stored through constraints in R_Z). Lastly, but not less important, several tools are available to deal with LTL specifications. In our experiments, the verifier chosen to compute refinement checks is *NuXMV*[16].

An efficient implementation of the CSCL algorithm has been developed using the encoding described in Section 6.3.2. Additionally, we decided to modify the CSCL algorithm to exploit multiprocessor architectures and further speed up synthesis. Algorithm 5, first computes a candidate solution and then it asks the verifier to validate or discard that candidate. The verifier execution is, in general, a time-consuming operation, i.e., verifying the validity of an LTL formula is a PSPACE-complete problem [68, 80]. We can observe, however, that it is possible (and convenient) to interrogate several verifier instances at the same time, providing them with different candidates. Here, we modify CSCL algorithm following this intuition, i.e., rejecting a candidate as soon as it is given to the verifier, and providing the ability to retrieve an *old* candidate in case one of the many verifier instances gives a positive answer. Algorithm 7 illustrates the parallel version of the CSCL algorithm. The Parallel CSCL algorithm is equivalent to the CSCL algorithm, as the two algorithms perform the same operations, with the difference that the Parallel CSCL generates candidates continuously, stopping only when a certain verifier instance indicates a successful candidate. The implementation of Parallel CSCL resulted in a tool we call PYCO[5].

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

- Minimize the number of components in H:

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

[5]https://github.com/ianno/pyco/releases/tag/SCP2018

```
1  function Parallel CSCL:
       Input: specification $S = (I_S, O_S, \delta_S, \sigma_S, R_S)$, library of components
              $L = (Z, T, R_Z, f)$, maximum number of components in the solution $N$.
       Output: set of components $H = \{G_1, \dots, G_n\}, H \subseteq Z$ and $n \leq N$, connection
               function $\theta$ such that $(\delta_1 \parallel \cdots \parallel \delta_n)\theta \sqsubseteq \delta_S$ and $f$ in minimized, or False if no
               solution is found.
2      initialize constraint solver and verifier, instantiating synthesis constraints for problem
        instance and setting $N$ as the maximum number of components in a candidate
        solution;
3      while get candidate solution $(H' = \{G_1, \dots, G_n\}, \theta')$ from constraint solver such
        that $f$ is minimized and all the synthesis constraints hold do
4          build composition $(G_1 \parallel \cdots \parallel G_n)\theta'$;
5          spawn a new verifier instance (process) to verify $(\delta_1 \parallel \cdots \parallel \delta_n)\theta' \sqsubseteq \delta_S$;
6          if any verifier instance has signaled success then
7              retrieve instance candidate $(H', \theta')$;
8              $H \leftarrow H'$;
9              $\theta \leftarrow \theta'$;
10             return $(H, \theta)$;
11         else
12             $R \leftarrow$ RejectCandidate$(H', \theta')$;   // infer equivalent erroneous
                candidates, see Algorithm 6
13             foreach $(H_t, \theta_t) \in R$ do
14                 add constraint $(H, \theta) \neq (H_t, \theta_t)$ to constraint solver;
15             end
16         end
17     end
18     wait for all the remaining running verifier instances to terminate;
19     if any verifier instance has signaled success then
20         retrieve instance candidate $(H', \theta')$;
21         $H \leftarrow H'$;
22         $\theta \leftarrow \theta'$;
23         return $(H, \theta)$;
24     end
25     return False;
26 end
```

Algorithm 7: Parallel CSCL algorithm.

- Minimize the number of ports used in the solution:

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

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

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

6.5 Case studies

This section contains two examples which illustrate the capability of the tool we developed, and, in general, of the `CSCL` algorithm. The goal of the first example is to design the control logic of a Brushless DC electric Motor (BLDC) Driver. The specification describes the waveform of the current used to control the electromagnets of the motor, and the task of the synthesizer is to figure out what components are necessary and how to connect them to ensure its proper operation. The second example, more challenging, requires the synthesizer to provide the control logic for the controller of an aircraft Electrical Power System (EPS), described in Chapter 4, where the specification describes some strict safety requirements that need to be always satisfied.

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

We ran all the experiments on a 3.3 GHz Intel Xeon machine, with 32GB of RAM, limiting the maximum number of parallel processes to 8. In some cases, however, we ran some experiments using the single process `CSCL` algorithm in Table 5.

6.5.1 The Brushless DC electric Motor Design (BLDC)

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

Brushless DC electric motors, as the name suggests, overcome those limitations by not having rotating brushes. In these motors, often the electromagnets are located on the stator while magnets are on the rotor, and the change in polarity is handled by a computer through high-power transistors [36]. BLDC motors are more precise, efficient, and have better performance than regular DC motors. They are more complex, however, as they require more electronic components to work properly. Although BLDC motors can be one, two, or three-phase, most of them are usually the latter type.

One of the simplest motor drivers for three-phase BLDC motors is the so-called *half bridge* configuration. Half bridges have this name because they only support the positive polarization of the electromagnets (instead of both negative and positive), generating only half of the maximum torque. Figure 6.4 shows the typical half bridge driver configuration, while Figure 6.5 illustrates the current waveforms required for the proper operation of the motor.

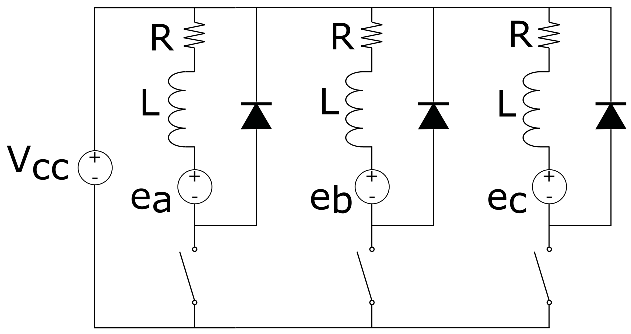


Figure 6.4: Common BLDC half bridge motor driver topology. Variables e_a, e_b, and e_c represent the Electromotive Force (EMF) for the three phases of the motor [36].

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

The goal of this example is to synthesize an architecture of components which is able to drive a simple BLDC motor[6]. In doing so, we want to show how our tool is able to infer a number of necessary components, correctly satisfying the specification both semantically (i.e., the A/G contract of the composition refines the A/G contract of the specification), and topologically (i.e., all the port types match). In this example, all the variables used in A/G contracts are Boolean. Table 6.1 illustrates the specification used for this case study.

The specification describes the interface of the motor driver, including one input and three outputs, at a logic level, without taking into account other physical constraints. The input, i, communicates to the driver whether the motor has reached a commutation point. The three outputs are signals which drive the current of the three phases of the motor. Given

[6]This example has been developed with Richard Lin and Rohit Ramesh.

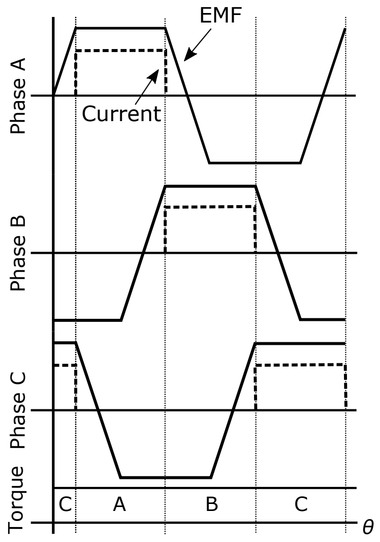


Figure 6.5: Waveforms for the half bridge driver. Input current from the driver induces torque through one of the three phases of the motor at a time [36]. To ensure the proper forward rotation of the rotor, inputs from the driver need to be sent in a specific order.

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 6.1: 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.

the input, the specification only requires that, when a commutation point is detected, only one driver signal is sent to the motor. The task of the synthesizer is to choose components from the library of 18 components, described in Table 6.2, and properly connect them to satisfy the specification.

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

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

Table 6.2: Structure of the BLDC library, which contains three separate instances of each component. Some components are architectural, meaning that they provide a typed interface but their A/G contract is always satisfied, and some are logic, providing both a typed interface and a non-trivial A/G contract (such as the MCU).

control logic required to satisfy the specification. Their port types, however, mismatch. The outputs of *MCU*, indeed, have type *IOPin3V*, while the specification requires outputs with type *IOPin12V*. Thus, Equation6.13 prevents their direct connection. It is responsibility of the synthesizer to figure out the right connections to propagate the control logic to ports of the right type.

We asked the synthesizer to find a solution using a constant cost function, minimizing the number of components in the solution, and minimizing the total number of ports. Additionally, we also ran a series of experiments using the single process `CSCL` algorithm in Algorithm 5, with a constant cost function. For each case, we ran 100 experiments summarized in Figure 6.6.

Interestingly, the overall fastest set of experiments was the one in which we minimized the number of components. Conversely, minimizing the number of ports led to the slowest synthesis times. Although the performance of constraints solvers is, in general, non-deterministic, we can explain these results by observing that the MCU component, although necessary to satisfy the specification, is the one with most ports. To minimize the number of ports used, the synthesizer tries avoiding it, without success, leading to a longer synthesis time. All the experiments resulted in correct designs, where the typical configuration was the set of components {Power-12V, DCDC-3V, MCU, Half-Bridge, Half-Bridge1, Half-Bridge2}, correctly connected. Figure 6.7 illustrates the typical solution for this case study. In the figure, the arrows between ports represent the function θ, defining port connections. An arrow between two ports, say a and b, means $\theta(a, b) = 1$.

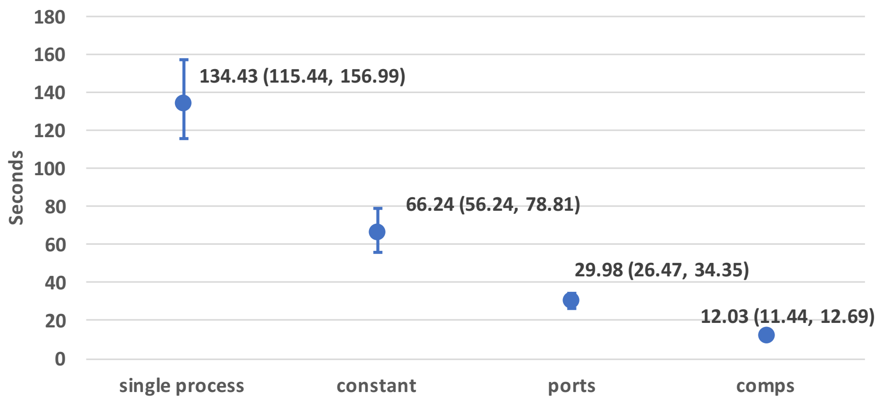


Figure 6.6: Summary of the results for the BLDC experiments. We synthesized a controller using the single process CSCL algorithm first, with a constant cost function. Then, we used the `Parallel CSCL` algorithm with a constant cost function, minimizing the number of components, and minimizing the number of ports. For each category, we ran 100 experiments. Each point represents the mean synthesis time, while the bars represent its 95% confidence interval, also indicated within parentheses.

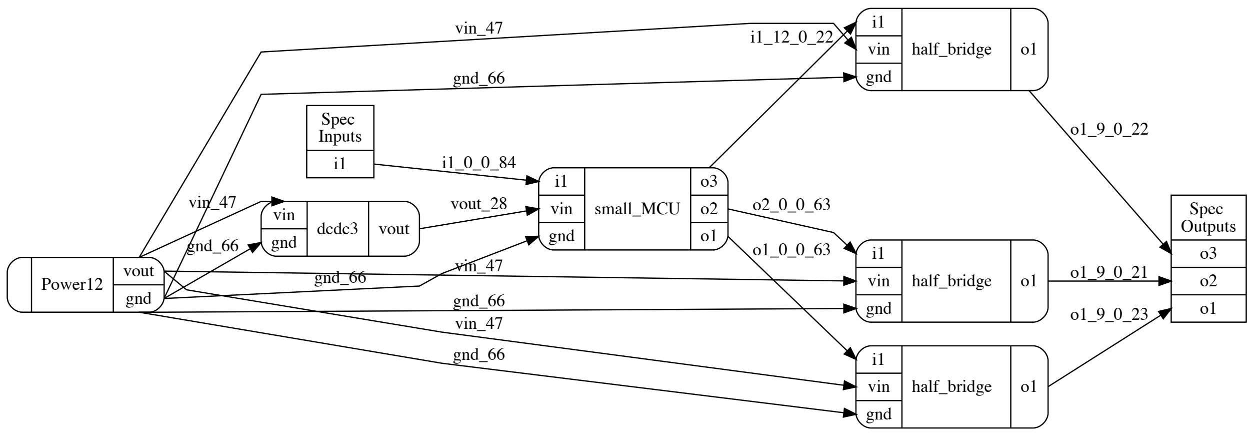


Figure 6.7: Graphical representation of a synthesized design. Names on arrows represent port renamings induced by the function θ, returned by the synthesis process.

6.5.2 The Aircraft Electrical Power System (EPS)

In this section, we present the result of the synthesis of a control unit, called *Bus Power Control Unit* (BPCU), for the EPS case study discussed in Chapter 4. Our goal is to synthesize the logic of the BCPU from a set of subsystem controllers, described by a library of A/G contracts. 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. The component library is the one described in Table 4.2.

We ran two series of experiments, one using a library containing 20 components, and one with 40 components. In both cases, the goal was to synthesize the BPCU according to the specifications in Table 4.1. For each series, we asked the synthesizer to find a solution first using the single process CSCL in Algorithm 5 with a constant cost function, and then using the parallel version in Algorithm 7 with a constant cost function minimizing the number of components used, and minimizing the number of ports in the solution. With both libraries, trying to minimize the number of components in the solution led to very long synthesis times, beyond the timeout that we set at 200 seconds.

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

Figure 6.8 shows the observed results, in terms of execution time, for the remaining cases. Interestingly, in the case of minimization of the number of ports, the synthesizer was able to find a solution generally faster than in the case of a constant cost function. In each graph, the dot represents the average synthesis time for one of the specification subsets $\{S_1\}, \{S_1, S_2\}, \ldots, \{S_1, \ldots, S_9\}$. For each of these subsets, we ran 100 experiments reporting their mean values, together with their 95% confidence intervals, in Table 6.3. As expected, one can immediately see how the parallel approach to synthesis is indeed more efficient than the single process one, with up to 50% performance improvement.

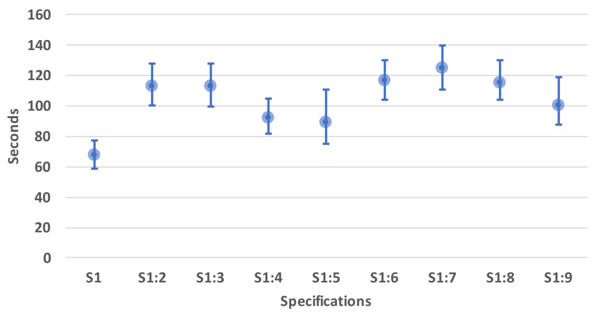
	Single Process 20	**Constant Cost 20**	**Minimize Ports 20**	**Single Process 40**	**Constant Cost 40**	**Minimize Ports 40**
$\{S_1\}$	67.43 (58.49, 76.95)	34.69 (30.85, 38.44)	10.69 (10.42, 10.98)	55.92 (46.44, 67.97)	28.13 (24.25, 34.40)	31.17 (30.07, 32.42)
$\{S_1, S_2\}$	113.12 (99.99, 127.91)	62.71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

Table 6.3: Summary of the EPS experiments. For each specification subset (one for each row), we report the mean value and its 95% confidence interval. All values are expressed in seconds. Experiments named "Constant Cost" and "Minimize Ports" are run using parallel processes.

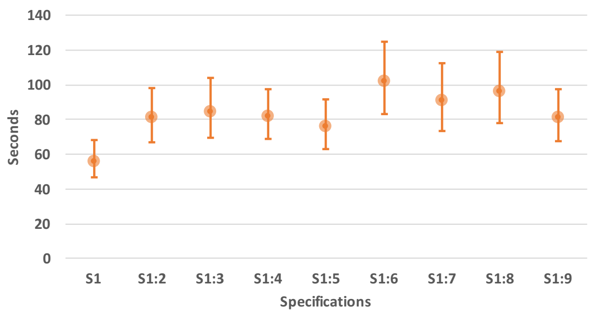
For reference, a typical solution satisfying all 9 specifications with the minimal number of connected ports included 5 components, $\{I_1, D_1, L_1, G_1, D_2\}$, for a total of 19 ports connected accordingly. Figure 6.9 represents the connections among the components according to the renaming function θ.

In a separate experiment, using the library with 40 elements, PYCO was able to explore the whole search space invoking the verifier 108176 times. This corresponded to more than 400M rejected candidates, which did not require an explicit check thanks to the inductive learning process described in Section 6.3. The verifier found a solution satisfying the specifications 386 times, corresponding to roughly 1.5M equivalent ones in the search space.

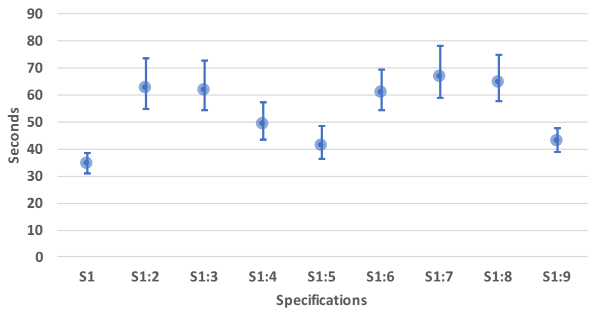
Figure 6.10 shows, instead, the effect of designer hints and library-specific constraints on synthesis time. Here synthesis is performed on smaller and simplified instances of the



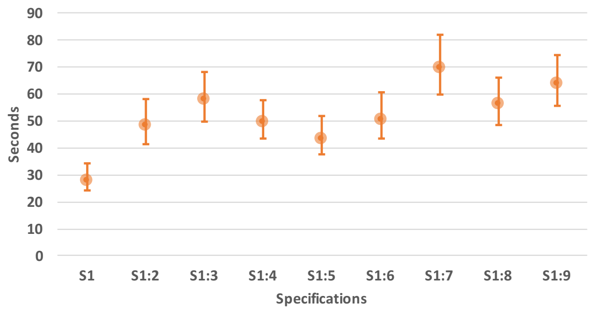
(a) Single process, constant cost, 20 elements.



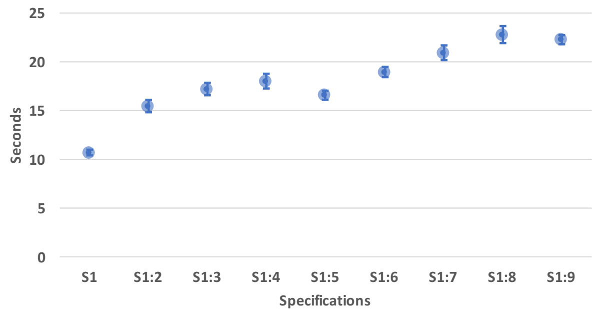
(b) Single process, constant cost, 40 elements.



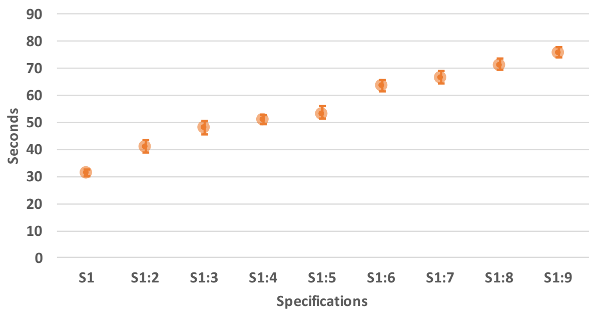
(c) Parallel execution, constant cost, 20 elements.



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



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



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

Figure 6.8: BPCU synthesis times in the various experiments. In each graph, each point has been computed running 100 experiments. The central point represents the mean, while bars represent the 95% confidence interval for the mean. The horizontal axis refers to the subset of specifications considered in each experiment.

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

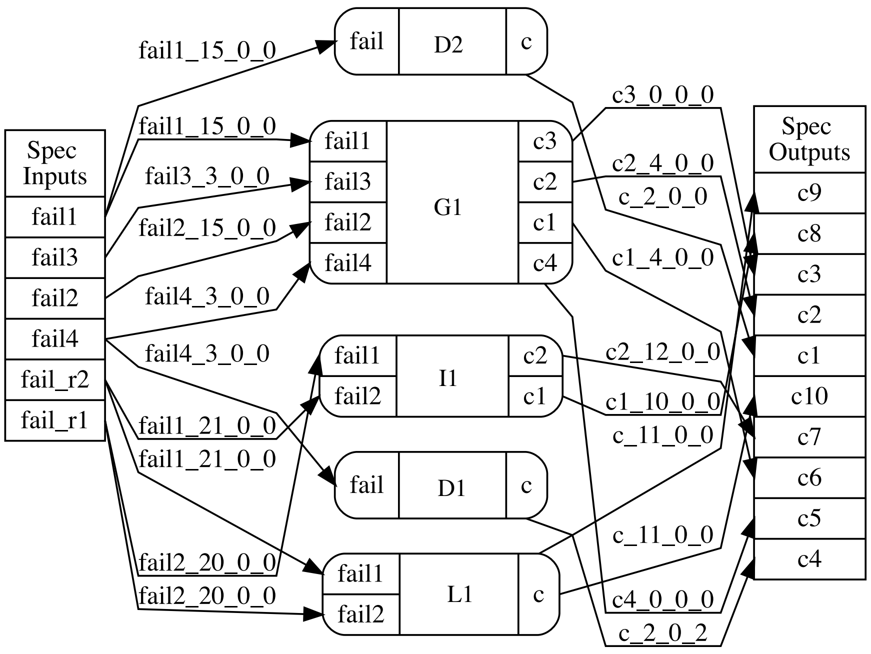


Figure 6.9: Graphical representation of a synthesized design satisfying all 9 specifications with minimal number of connected ports. Names on arrows represent port renamings induced by the function θ, returned by the synthesis process.

6.6 Conclusion

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

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

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

In the next chapters, instead, we will analyze the same problem of constrained synthesis

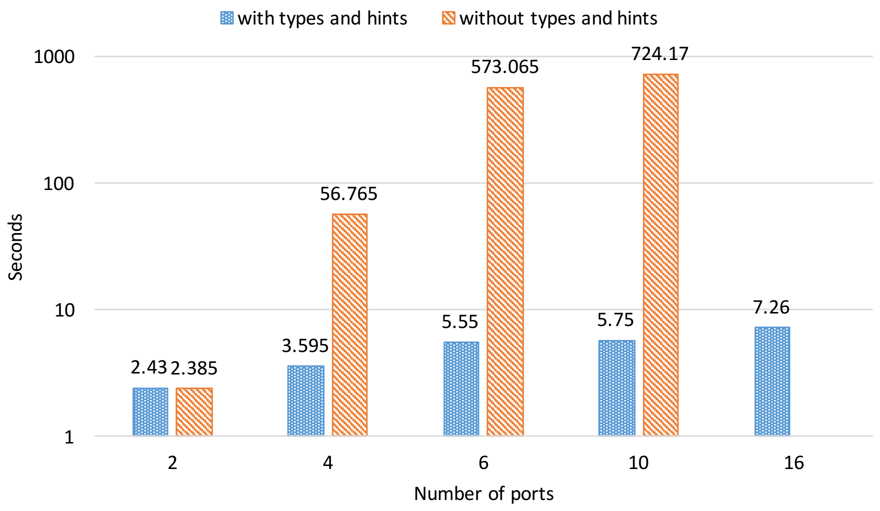


Figure 6.10: Impact of types and user provided hints on synthesis time for simplified instances of the EPS example. Each bar of the histogram represents the median value from 10 experiments. The graph is in logarithmic scale. In the case without types and 16 ports, most of the times the synthesizer has not been able to find a solution within the time limit of 1000 seconds.

from component libraries relaxing the requirement of the verifier only providing a yes/no answer. Additionally, we will focus exclusively on LTL A.G contracts, while here we considered only generic components, and show how we can leverage their properties to make synthesis more efficient.

Chapter 7

Synthesis from Libraries of LTL A/G Contracts

In this chapter, we focus on a very similar problem of Chapter 6, i.e., we want to synthesize a composition of elements from a library that satisfies a certain specification. In the other chapter, however, we assumed the components were generic and that our verifier was only able to answer yes or no when asked whether a candidate solution was indeed the correct one. Here, instead, we frame the problem in terms of LTL A/G Contracts, and we explicitly use a model checker to validate candidates solutions. We propose a CEGIS-based approach which leverages the infinite-length counterexamples generated by the model checker, represented compactly as state machines, to guide the synthesis process. Compared to Chapter 6, this synthesis strategy is more efficient as it requires fewer calls to the verifier, although it requires more work to generate candidate solutions. We evaluate our algorithm by thy synthesizing a controller in three case studies.

The rest of the chapter is organized as follows. After short introduction in Section 7.1, in Section 7.2 we present the synthesis problem that we address in this chapter. We propose a solution for a simplified version of the problem in Section 7.3. In Section 7.4 we focus on performance issues, describing, then, our approach to the full problem in Section 7.5. We discuss the detail of our CEGIS algorithm in Section 7.6. In Section 7.7 we applying the synthesis technique to several case studies, and we conclude in Section 7.8.

7.1 Introduction

In Chapter 6, we focused on the problem of synthesis from libraries of components when no assumptions on the formalism used to describe components are made, and assuming minimal additional information from the verifier. The only requirements we considered were being able of composing components, and, given two different components, being able to determine if one is the refinement of the other, i.e., it can be seen as an implementation of the latter. Hence, we only required the verifier to provide a yes/no answer.

As discussed, that synthesis approach relies on the generation of many candidate solutions which can be independently verified. Specifically, the size of each verification problem only depends on the specification itself and the candidate solution, and it does not grow with the number of tested candidates. In our experiments, we used LTL A/G contracts to describe components as a particular instance of the framework, and discussed other possible types of component specifications. Performance is acceptable when the library specifies a large number of constraints, limiting the total number of possible candidates. When the size of the library increases, however, scalability remains an issue.

In this chapter, instead, we look at a similar problem, focusing solely on LTL A/G contracts and the ability of a model checker of returning counterexample traces. Thus, here we talk about libraries of contracts. This choice allows us to leverage properties that are unique to LTL and A/G contracts, without limiting ourselves to only observing the Boolean outcome of the refinement checking process. Hence, we can access more detailed information provided by the underlying model checker, i.e., counterexamples representing input sequences, and use them to improve synthesis' efficiency.

Compared to traditional CEGIS approaches [81, 34, 49], however, in this case, we need to tackle several additional technical challenges. For instance, using LTL, our inputs and outputs are infinite temporal sequences. What is the best way to represent them? How to ensure termination of the algorithms, given that the input space is infinite? We will answer these questions later in the next sections.

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

The problem we want to solve is a variation the one in Definition 9:

Definition 10 (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 $\mathcal{R}$ includes also 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*

$$(C_1 \otimes \cdots \otimes C_N)\theta \preceq S \tag{7.1}$$

where well-defined is intended as in Definition 2. With respect to the problem in Definition 9, here we leverage definitions and notations that are proper of our contract framework, but the ultimate goal is substantially the same, 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 superset containing all the legal connections for L.

7.3 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 $H = \mathcal{Z}$. 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 for a generic $H \subseteq \mathcal{Z}$.

Our solution follows the CEGIS paradigm, under the assumption that we will be able to obtain counterexamples over the system variables from the verifier.

As in Chapter 6, we will indicate all the ports in the library as $\mathcal{P}_{lib}$, defined in Equation 6.1, and all the ports in the library and the specification as $\mathcal{P}_{lib \cup S}$, defined in Equation 6.2.

7.3.1 LTL-CSCL as a CEGIS instance

To encode possible connections between contracts, we use the approach introduced in Section 6.3.2, although here all the definitions are applied here to contracts rather than generic components. Thus, each port in $\mathcal{P}_{lib \cup S}$ has an associated index and connection variable, defined according the functions $\mathcal{I}$ and $\mathcal{M}$, respectively. For a certain θ, for instance, we can assert the connection $(x, y) \in \theta$ as $\mathcal{M}(x) = \mathcal{I}(y)$. If the connections implied by $\mathcal{M}$ represent the right interconnect for a library L and specification S, we write $\mathcal{M} \models_L S$, otherwise we write $\mathcal{M} \not\models_L S$. From the assignments defined through $\mathcal{M}$, one can immediately derive the implied interconnect θ. Note, however, that the connections we can represent with $\mathcal{M}$, however, are fewer than those we can express with θ (for instance, we cannot connect two outputs of contracts in the library). Thus, given a $\mathcal{M}$, we indicate its derived interconnect as $\theta_{\mathcal{M}}$. Given a library $L = (\mathcal{Z}, \mathcal{R})$, we can express all the legal connections among contracts, according to $\mathcal{R}$, in relation to their index using the following LTL formulas.

$$\phi_{\mathcal{M}L} = \bigwedge_{C=(I,O,\varphi,\psi)\in Z} \; \bigwedge_{p \in I} \left[\bigvee_{(p,q)\in\mathcal{R}} [\mathcal{M}(p) = \mathcal{I}(q)] \rightarrow \Box(p = q) \right] \tag{7.2}$$

$$\phi_{\mathcal{M}S} = \bigwedge_{s \in O_S} \; \bigvee_{(s,q)\in\mathcal{R}} \{[\mathcal{M}(s) = \mathcal{I}(q)] \rightarrow \Box(s = q)\} \tag{7.3}$$

Equation 7.2 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, $\mathcal{M}(x) = \mathcal{I}(y)$, then we also expect the formula $\Box(x = y)$ to hold, meaning that x and y are connected and they have to expose 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.

Equation 7.3 encodes the connections of the output ports of the specification, and it is very similar to Equation 7.2. We refer to the conjunction of the two formulas simply as

$$\phi_{\mathcal{M}} = \phi_{\mathcal{M}L} \wedge \phi_{\mathcal{M}S} \tag{7.4}$$

Equation 7.4 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 useful defining a contract representing the composition of all contracts in the library:

$$C_L = (I_L, O_L, \varphi_L, \psi_L) = \bigotimes\{C_i \mid C_i \in \mathcal{Z}\} \tag{7.5}$$

The contract C_L, while convenient to represent the whole library as a single composition, does not tell us anything on the connections among its constituent contracts. Our goal is to find a renaming $\mathcal{M}$ according to:

$$\exists \mathcal{M} \colon \phi_{\mathcal{M}} \Rightarrow (C_L \preceq S) \tag{7.6}$$

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

Considering the constraint implied by Equation 7.2—no input port of contracts in the library can be unconnected—together with Equation 7.3, which requires all the specification outputs to be mapped to contracts in the library, we have that Equations 3.22a and 3.22b (part of the refinement conditions) will always be satisfied. Equation 7.6, then, can be written as $\mathcal{M}$:

$$\exists \mathcal{M} \colon \phi_{\mathcal{M}} \rightarrow (\varphi_S \rightarrow \varphi_{C_L}) \wedge (\psi_{C_L} \wedge \rightarrow \psi_S) \text{ is valid} \tag{7.7}$$

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

$$\exists \mathcal{M} \colon \forall \sigma \colon \sigma \models [\phi_{\mathcal{M}} \rightarrow (\varphi_S \rightarrow \varphi_{C_L}) \wedge (\psi_{C_L} \wedge \rightarrow \psi_S)] \tag{7.8}$$

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

Equation 7.8, which represents in a concise form the problem described in Definition 10, is typical of CEGIS problems, i.e. in the $\exists\forall$ form. As in [48], 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).

```
function General LTL-CSCL:
    Input: library contract $C_L = (I_L, O_L, \varphi_L, \psi_L)$, constraint formula $\phi_{\mathcal{M}}$, specification
           contract $S = (I_S, O_S, \varphi_S, \psi_S)$
    Output: set of connections representing $\theta$, or False
    $\mathcal{E} \leftarrow \{True\}$;                                  // Counterexample set
    while True do
        model ← checkSAT($\phi_{\mathtt{syn}}$($C_L, S, \phi_{\mathcal{M}}, \mathcal{E}$));
        if model is unsat then
            return False;
        end
        $\theta_{\mathcal{M}}$ ← extractCandidate(model);
        model ← checkValid($\phi_{\mathtt{ver}}$($C_L, S, \theta_{\mathcal{M}}$));
        if model is valid then
            return c;
        end
        add counterexample from model to $\mathcal{E}$;
    end
end
```

Algorithm 8: 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.

7.3.2 Implementation of the CEGIS Paradigm for the LTL-CSCL problem

Algorithm 8 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 4, we check for the satisfiability of a synthesis constraint, i.e., $\phi_{\mathtt{syn}}$, which we will derive from Equation 7.8 and the list of counterexamples seen until that point. This operation returns a model, from which we can extract a candidate set of port connections $\theta_{\mathcal{M}}$ (line 8), or `unsat` if no candidate is found. Then, in line 9, the candidate set `c` is used to derive a different constraint, $\phi_{\mathtt{ver}}$. If $\phi_{\mathtt{ver}}$ is valid, then `c` represents the correct set of assignments to be encoded as θ. Otherwise, the `checkValid` routine returns a trace that describes a counterexample for the validity check, which is added the list of counterexamples $\mathcal{E}$. 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 Section 3.5.

Although describing the essence of our approach to solve the LTL-CSCL problem, Algorithm 8 is still not complete. For instance, we have no guarantee that the synthesis loop will ever terminate, and the best representation of the sequences σ of Equation 7.8 derived from the counterexamples is still undefined. Additionally, we still have to fully describe the constraints $\phi_{\texttt{syn}}$ and $\phi_{\texttt{ver}}$. In the next sections, we will discuss these issues and provide the missing details of our solution.

7.3.3 Handling Infinite Input Sequences

The first problem we need to address is to guarantee that Algorithm 8 terminates. In [81], 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 valid. 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 such that $C_L\theta_i \not\preceq S$. Then, we can express a constraint which enumerates all the wrong candidates and prevents them to appear again as:

$$\phi_{\mathcal{W}} = \neg \bigvee_{\theta_i \in \mathcal{W}} \bigwedge_{(x,y)\in\theta_i} (\Box(x = y)) \tag{7.9}$$

To prevent the generation of old candidates, then, will need to consider Equation 7.9 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 very large.

The other problem that arises from having infinite sequences as counterexamples is finding a suitable way to represent them. In Equation 7.8, 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, as explained in Section 3.5, 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 satifiability of a formula $\forall\sigma\colon \sigma \models \phi$, or simply ϕ, we need to model check:

$$\mathcal{T} \subseteq \mathcal{L}(\phi) \text{ for validity} \tag{7.10a}$$

$$\mathcal{T} \subseteq \mathcal{L}(\neg\phi) \text{ for satisfiability} \tag{7.10b}$$

Equations 7.10 helps us understand how to relate traces and LTL formulas, and can be checked by a model checker. In Section 3.5.2, we discuss how all counterexamples generated by a model checker (NuXMV in our case) are either finite or lazo-shaped. For such counterexamples, we can always find a corresponding finite state machine that is able to produce the specific sequence they describe.

```
function TraceGenerator:
    Input: counterexample trace $c$, set of variables $\mathcal{V}$
    Output: state machine $M$ generating trace $c$
    $S$ : list;                                   // list of states
    $T$ : list;                                   // list of transitions
    $D$ : hashtable;          // map of variable evaluations for each state
    $n \leftarrow$ numberOfStates($c$);
    $l \leftarrow$ loopIndex($c$);        // $l \in \{-1\} \cup \{1, \cdots, n\}$.  $-1$ means no loop
    $i \leftarrow 1$;
    $S \leftarrow$ add initial state $i$;
    $D[i] \leftarrow$ evaluateTraceAt($\mathcal{V}$,$c$,$i$);
    for $i \in \{1, \cdots, n\}$ do
        $S \leftarrow$ add state $i$;
        $D[i] \leftarrow$ evaluateTraceAt($\mathcal{V}$,$c$,$i$);
        $T \leftarrow$ add transition from state $i$ $-1$ to $i$;
        if $i = n$ then
            if $l = -1$ then // we add a generic state with no evaluation
                $S \leftarrow$ add state $i + 1$;
                $T \leftarrow$ add transition from state $i$ to $i + 1$;
                $T \leftarrow$ add transition from state $i + 1$ to $i + 1$;
            else // there is a loop, we go back to that state
                $T \leftarrow$ add transition from state $i$ to $i + 1$;
            end
        end
    end
    return $M = (S, D, T)$
end
```

Algorithm 9: Constructs a finite state machine able to generate the same sequence of symbols described by a counterexample trace, as described in Section 3.5.2. The resulting state machine can then immediately be encoded as an SMV module.

Algorithm 9 describes 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 [56].

At this point, we have all the elements to derive an initial version of the two formulas ϕ_{syn} and $\phi_{\mathtt{ver}}$ introduced in Algorithm 8. As we will revisit these formulas later, we refer to them here as ϕ'_{syn} and $\phi'_{\mathtt{ver}}$. In ϕ'_{syn}, we want to find one assignment for $\mathcal{M}$ which satisfies all the counterexamples seen until a certain iteration. While in Equation 7.8 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 3.2.1.5, that we expect the correct behavior of the contract under correct assumptions. Thus, ϕ'_{syn} becomes:

$$\phi'_{\mathrm{syn}} = \exists \mathcal{M} \colon \forall \sigma_i \in \mathcal{E} \colon \{\sigma_i \models [\phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge (\varphi_S \wedge \psi_S) \wedge (\varphi_{C_L} \wedge \psi_{C_L})]\}[x/x_i]_{x \in \mathcal{P}_{lib \cup S}} \tag{7.11}$$

where $\mathcal{E}$ is the set containing the observed counterexamples introduced in Algorithm 8, and $[x/x_i]_{x \in \mathcal{P}_{lib \cup S}}$ 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:

$$\bigwedge_{\sigma_i \in \mathcal{E}} \{\mathcal{L}(F_{\sigma i}) \subseteq \mathcal{L}(\neg[\phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge (\varphi_S \wedge \psi_S) \wedge (\varphi_{C_L} \wedge \psi_{C_L})])\}[x/x_i]_{x \in \mathcal{P}_{lib \cup S}} \tag{7.12}$$

where $F_{\sigma i}$ is a state machine generated from σ_i according to Algorithm 9. 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 7.2) are not renamed across the multiple 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 $\phi'_{\mathtt{ver}}$, instead, we are interested in proving that the candidate $\theta_{\mathcal{M}}$, derived by the assignments of $\mathcal{M}$ in ϕ'_{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:

$$\phi'_{\mathtt{ver}} = C_L\theta_{\mathcal{M}} \preceq S = (\varphi_S \rightarrow \varphi_{C_L\theta_{\mathcal{M}}}) \wedge (\psi_{C_L\theta_{\mathcal{M}}} \rightarrow \psi_S) \tag{7.13}$$

Although, at this point, we can say that using $\phi'_{\mathtt{syn}}$ and $\phi'_{\mathtt{ver}}$ as synthesis and verification constraints in Algorithm 8 will yield a correct result, 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.

7.4 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 $\varphi_S \neq \psi_S \neq \mathit{True}$, also with p input and output ports, and try to observe how the CEGIS Algorithm 8 would work in such case, with $\phi'_{\mathtt{syn}}$ and $\phi'_{\mathtt{ver}}$ as defined in the previous section. First, in line 4, the algorithm tries to satisfy Equation 7.11, 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 7.13, in line 9, 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 incredibly slowly, as for each counterexample the formula in Equation 7.12 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.

Algorithm 10 shows a procedure that can detect whether a contract C is nondeterministic, meaning that, for a certain sequence over its input ports which 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 4 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 4 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

```
function IsDeterministic:
    Input: contract $C = (I, O, \varphi, \psi)$
    Output: True if $C$ is deterministic, False otherwise
    $C' \rightarrow$ copy of $C$ for interconnect $\theta$;        // as defined in Section 3.2.1.6
    $\rho_I \rightarrow \bigwedge_{i \in I} \square(i = \bar{\theta}(i))$;        // for $\bar{\theta}$ see Equation 3.17
    $\rho_O \rightarrow \bigwedge_{o \in O} \square(o = \bar{\theta}(o))$;
    m = checkValid($(\varphi \wedge \psi) \wedge (\varphi' \wedge \psi') \wedge \rho_I \rightarrow \rho_O$);
    if m is valid then
        return True
    else
        return False
    end
end
```

Algorithm 10: 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. $\bar{\theta}$ is a function returning the variable associated to the function parameter according to θ.

our synthesis algorithm, but can be used while defining the contract library to make sure all the contracts have deterministic behaviors.

7.4.1 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 8 (Simple Cycle)**.** *Let $C = (\{a\}, \{b\}, True, \square(b = a))$ be a contract which 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 8 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 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 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

$$\mathcal{V}_L = \{l_C \mid C \in \mathcal{Z}\} \tag{7.14}$$

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:

$$\phi_{LL} = \bigwedge_{C_1=(I_1,O_1,\varphi_1,\psi_1)\in Z} \ \bigwedge_{p\in I_1} \ \bigwedge_{C_2=(I_2,O_2,\varphi_2,\psi_2)\in Z} \ \bigwedge_{q\in O_2} [\mathcal{M}(p) = \mathcal{I}(q) \rightarrow (l_{C1} > l_{C2})] \tag{7.15}$$

and greater than zero if mapped to the specification inputs:

$$\phi_{LS} = \bigwedge_{C_1=(I_1,O_1,\varphi_1,\psi_1)\in Z} \ \bigwedge_{p\in I_1} \bigwedge_{q\in O_S} [\mathcal{M}(p) = \mathcal{I}(q) \rightarrow (l_{C1} > 0)] \tag{7.16}$$

Then, we indicate as ϕ_L the conjunction of Equations 7.15 and 7.16:

$$\phi_L = \phi_{LL} \wedge \phi_{LS} \tag{7.17}$$

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 $\mathcal{V}_L$:

$$\phi_D(d) = \bigwedge_{v\in\mathcal{V}_L} v \leq d \tag{7.18}$$

where d is the desired depth. Moreover, given a maximum desired 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 8.

7.5 Addressing the Full LTL-CSCL Problem

In this section, we will relax the assumption $H = \mathcal{Z}$ that we made in Section 7.3, 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 9 (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

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

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

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

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 9 shows why, sometimes, requiring a solution to a synthesis problem to include all the contracts in the library is not a good idea. One approach could be the one discussed in Chapter 6, where a constraint solver is used to pre-select sets of components which are then tested for refinement. Used in this chapter's context, however, that approach would not yield any additional benefit, as it would result in a synthesis algorithm that is no better than Algorithm 5, which tests many candidate solutions only once.

Another solution, which we prefer in this case, is to give the ability to the synthesis engine, i.e., the model checker, to select which contracts to include in a solution within the main synthesis loop. We do so by using the contract location variables introduced in Equation 7.14, i.e., 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:

$$C(l_C) = (I, O, (l_C > 0) \rightarrow \varphi, (l_C > 0) \rightarrow \psi) \tag{7.19}$$

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 7.5) as

$$C_L(\mathcal{V}_L) = (I_L, O_L, \varphi_L(\mathcal{V}_L), \psi_L(\mathcal{V}_L)) = \bigotimes\{C_i(l_{Ci}) \mid C_i \in \mathcal{Z}\} \tag{7.20}$$

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

$$\phi_C = \bigwedge_{[C_1=(I_1,O_1,\varphi_1,\psi_1)\in Z]} \bigwedge_{[p\in I_1]} \bigwedge_{[C_2=(I_2,O_2,\varphi_2,\psi_2)\in Z]} \bigwedge_{[q\in O_2]} [l_{C2} = 0 \rightarrow \mathcal{M}(p) \neq \mathcal{I}(q)] \tag{7.21}$$

We enforce this additional constraint because, otherwise, we would allow nondeterministic inputs which are not desirable, as discussed in Section 7.4.1.

7.6 Putting It All Together: Revisited LTL-CSCL

In this section, we revisit the initial synthesis and verification constraints, $\phi'_{\mathtt{syn}}$ and $\phi'_{\mathtt{ver}}$, introduced in Equations 7.11 and 7.13, to include all the improvements discussed in Sections 7.4 and 7.5. We start from the synthesis constraint, which now becomes:

$$\begin{aligned}&\phi_{\mathtt{syn}}(C_L, S, \phi_{\mathcal{M}}, \mathcal{E}, \mathcal{W}, d) = \exists \mathcal{M}, \mathcal{V}_L \colon \forall \sigma_i \in \mathcal{E} \colon \\ &\qquad \{\sigma_i \models \phi_{\mathcal{M}} \wedge \phi_{\mathcal{W}} \wedge \phi_L \wedge \phi_D(d) \wedge \phi_C \wedge (\varphi_S \wedge \psi_S) \wedge (\varphi_{C_L}(\mathcal{V}_L) \wedge \psi_{C_L}(\mathcal{V}_L))\}[x/x_i]_{x\in\mathcal{P}_{lib\cup S}}\end{aligned} \tag{7.22}$$

where σ_i represents the sequence generated by the *ad-hoc* state machine built according to Algorithm 9. The formula above can be checked for satisfiability by solving a problem similar to the one in Equation 7.12. It is worth noticing how, this time, we are also looking for an assignment of the variables in $\mathcal{V}_L$, which parameterize the library contract C_L. The addition of all the extra terms, here, is not a problem. A candidate $\mathcal{M}$ satisfying Equation 7.22, if correct, will also satisfy Equation 7.8, which represents our synthesis goal.

The verification constraint, instead, does not change but 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 $\phi_{\mathtt{syn}}$. We indicate the candidate composition as $C_H = (I_H, O_H, \varphi_H, \psi_H) = \bigotimes\{C_i \mid C_i \in H\}$, and the verification constraint includes it as:

$$\phi_{\mathtt{ver}}(C_H, S, \theta_{\mathcal{M}}) = \phi'_{\mathtt{ver}}(C_H, S, \theta_{\mathcal{M}}) = (\varphi_S \rightarrow \varphi_{C_H\theta_{\mathcal{M}}}) \wedge (\psi_{C_H\theta_{\mathcal{M}}} \rightarrow \psi_S) \tag{7.23}$$

where $\theta_{\mathcal{M}}$ is the interconnect inferred from $\mathcal{M}$.

We can now provide a detailed representation of the LTL-CSCL algorithm, described in Algorithm 11, which takes into account the newly added constraints and the incremental depth strategy.

```
function Full_LTL-CSCL:
    Input: library contract $C_L = (I_L, O_L, \varphi_L, \psi_L)$, constraint formula $\phi_{\mathcal{M}}$, specification
           contract $S = (I_S, O_S, \varphi_S, \psi_S)$, maximum solution depth $D$
    Output: set of contracts $H \in \mathcal{Z}$ and set of connections representing $\theta$, or $False$
    $d = 1$;
    $\mathcal{E} \leftarrow \{\mathcal{T}\}$;                                  // Counterexample set
    $\mathcal{W} \leftarrow \{\}$;                                      // Old candidates
    while $d \leq D$ do
        while $True$ do
            model $\leftarrow$ checkSAT($\phi_{\text{syn}}(C_L, S, \phi_{\mathcal{M}}, \mathcal{E}, \mathcal{W}, d)$);
            if model is unsat then
                $d = d + 1$;
            end
            $(H, \theta) \leftarrow$ extractCandidate(model);
            $\mathcal{W} \leftarrow \mathcal{W} \cup \{(H, \theta)\}$;
            model $\leftarrow$ checkValid($\phi_{\text{ver}}(C_H, S, \theta)$);
            if model is valid then
                return $(H, \theta)$;
            end
            $\sigma \leftarrow$ extractCounterexample(model);
            $F_\sigma \leftarrow$ TraceGenerator($\sigma$);
            $\mathcal{E} \leftarrow \mathcal{E} \cup \{F_\sigma\}$;
        end
    end
    return $False$;
end
```

Algorithm 11: Description of the algorithm solving the LTL-CSCL problem including all the performance considerations discussed in Section 7.4.

7.7 Evaluation

In this section we discuss the evaluation of Algorithm 11 in three different scenarios. The first two are the already seen aircraft electrical power system (cf. Chapter 4) and brushless electric motor (cf. Section 6.5.1) synthesis problems. The third problem 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).

We implemented the algorithm in Python, as an extension of the tool Pyco introduced in Section 6.4. We rely on the NuXMV model checker to compute the validity and satisfiability of formulas, and to execute the state machines generating the counterexample traces. All the tests were executed on a 3.3 GHz Intel Xeon machine, with 32GB of RAM.

7.7.1 The BLDC problem

In this section we discuss the results of the `Full_LTL-CSCL` synthesis algorithm applied to the BLDC example introduced in Section 6.5.1. Before proceeding, however, we need to introduce a revisited version of the contract library that we used earlier.

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

Table 7.1: Structure of the BLDC library, which contains three separate instances of each component. This library is a revisited version of the one in Table 6.2, which now includes only deterministic contracts.

The library in Table 6.2, in fact, contains contracts that are nondeterministic, which in this case can lead to performance issues, as discussed in Section 7.4.1. The contract representing the microcontroller, MCU, is a problematic one. For instance, the contract does not specify what to do in case the input i is true for more than one cycle. Table 7.1 describe the updated library, which contains now only deterministic contracts. The identification of the nondeterministic contract and the validation of the updated library was performed using Algorithm 10.

Figure 7.1 shows the synthesis times, for libraries derived from Table 7.1 with 16, 24, and 32 contracts. Each point represents the bootstrap mean of an initial sample size of 80 experiments, while the bars indicate the 95% bootstrap confidence interval for the mean. In these experiments, the solver was able to find a solution almost immediately—most of the times it took only two iterations to find a good solution. This can explain how having a larger library, in this case, does not translate in exponential performance degradation (which, however, is expected for larger libraries). Synthesis using the nondeterministic library generated comparable results, although with much higher variance. For instance, for a certain

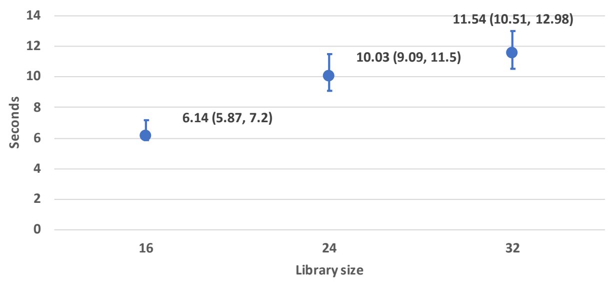


Figure 7.1: Summary of the results for the BLDC experiments. We synthesized a controller using three libraries including 16, 24, and 32 contracts respectively, for the same specification. For each category, we ran 80 experiments, and then bootstrapped the samples. Each point represents the bootstrap mean for the synthesis time, while the bars represent its 95% confidence interval, also indicated within parentheses.

library, it would not be uncommon to see some experiments taking more than five times the synthesis time of others. Resulting compositions were similar to the one in Figure 6.7

7.7.2 The EPS problem

Here we summarize the results of applying the `Full_LTL-CSCL` algorithm to the EPS case study (introduced in Chapter 4 and discussed further in Section 6.5.2). As for the BLDC case, also in this case a contract was found to be nondeterministic. To ensure determinism, in these experiments we replaced contract A_1 of Table 4.2 with another instance of contract B_1.

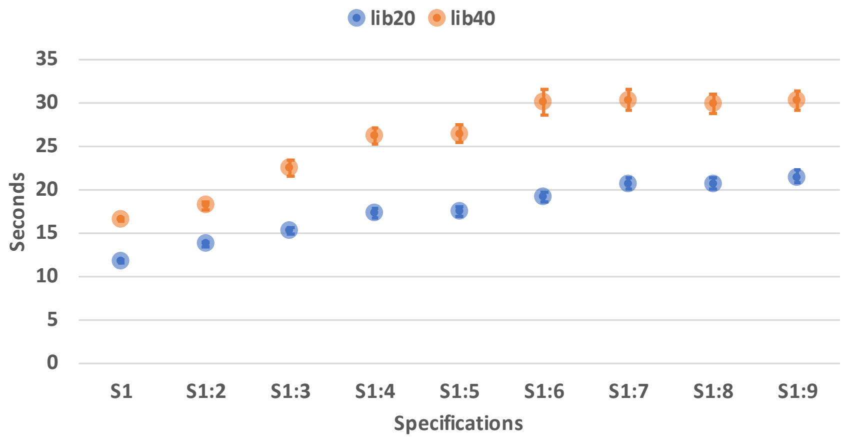


Figure 7.2: Summary of the EPS experiments. We synthesized a controller using two different libraries, one with 20 and the other with 40 contracts, defined in Table 4.2. For each category, we ran 80 experiments and then bootstrapped the sample. Each point represents the bootstrap mean synthesis time, while the bars represent its 95% bootstrap confidence interval.

	Library 20	**Library 40**
$\{S_1\}$	11.82 (11.66, 11.99)	16.57 (16.35, 16.77)
$\{S_1, S_2\}$	13.76 (13.49, 14.02)	18.14 (17.65, 18.68)
$\{S_1, \ldots, S_3\}$	15.26 (14.93, 15.65)	22.47 (21.55, 23.48)
$\{S_1, \ldots, S_4\}$	17.32 (16.81, 17.92)	26.13 (25.22, 27.15)
$\{S_1, \ldots, S_5\}$	17.41 (16.89, 17.99)	26.45 (25.46, 27.52)
$\{S_1, \ldots, S_6\}$	19.12 (18.54, 19.74)	30.01 (28.69, 31.5)
$\{S_1, \ldots, S_7\}$	20.7 (20.09, 21.39)	30.36 (29.19, 31.64)
$\{S_1, \ldots, S_8\}$	20.7 (20.15, 21.3)	29.87 (28.84, 31.05)
$\{S_1, \ldots, S_9\}$	21.45 (20.8, 22.25)	30.25 (29.21, 31.31)

Table 7.2: 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 value and its 95% confidence interval (in parentheses). All values are expressed in seconds.

We ran each experiment 80 times and then bootstrapped the samples to obtain mean and 95% bootstrap confidence interval. Figure 7.2 summarizes the results, which are reported in detail in Table 7.2. If compared to the results reported in Section 6.5.2 (cf. Table 6.3), we can observe that the approach presented here is roughly twice as fast, if compared to the constant cost case. Interestingly, while for the experiments in Section 6.5.2 we reported an experiment generating more than 100000 verification queries, here no experiment took more than 8 iterations to converge to a solution. Figure 6.9 illustrates a typical composition resulting from the synthesis procedure.

7.7.3 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 extremely useful 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 worlds. 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, [47]). The SPI protocol implements the master-slave architecture, where the two parties communicate through four signals, as illustrates in Figure 7.3.

In this section, we study a scenario which is affine to the technologies we just introduced[1]. 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 a parallel

[1]This example has been developed with Íñigo Íncer Romeo.

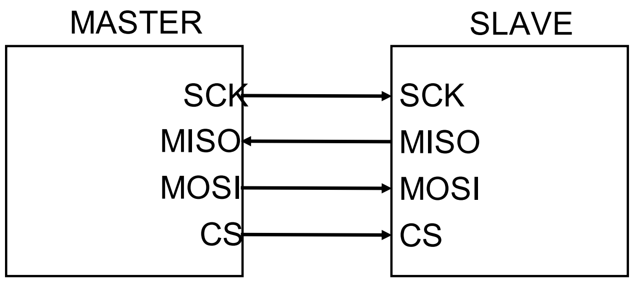


Figure 7.3: Synthesis times with and without specification decomposition. The horizontal axis indicates a subset of the guarantees in Table 4.1, while the vertical one, in logarithmic scale, the synthesis time.

fashion. Here, the specification does not implement the SPI 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 7.3 provides the details about the contract representing the specification.

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

Table 7.3: Specification for the SPI-ADC synthesis problem parameterized for n (in the experiments, $n \in \{2, 3, 4, 5\}$). 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 7.4 shows, instead, the elements that populate the contract library. The set of contracts is very simple. 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, it means that the analog value will not be converted at once, but each bit will be computed sequentially The rest of the library contains very simple 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 match their parameter, triggers,

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

Table 7.4: Structure of the ADC-SPI library. The ADC component is used, in the experiments, with $n \in \{2, 3, 4, 5\}$. 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.

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.

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 do more work to verify candidates.

Figure 7.4 shows the synthesis times for the specification for $n \in \{2, 3, 4, 5\}$, executed with libraries containing an ADC parameterized with the same n, for library with size of $11, 11, 16, 17$ contracts, respectively. Note, however, that most of the elements in the library are parametric, with a parameter $0 \leq k \leq 10$, meaning that each of those contracts encoded a larger search space. We can observe how the synthesizer is able to find solutions in a

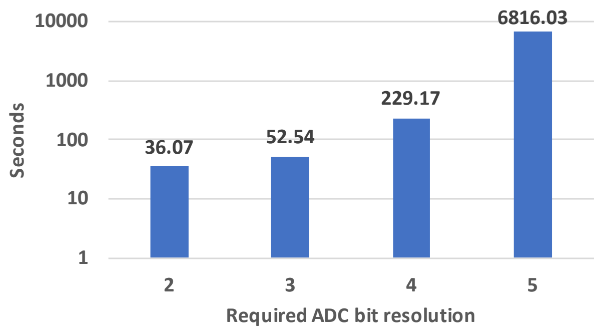


Figure 7.4: Summary of the results for the ADC-SPI experiments, for specifications with parameter $n \in \{2, 3, 4, 5\}$ and library size of 11, 11, 16, 17 contracts, respectively. Each library had the appropriate ADC according to n. The other contracts, including the parametric ones, were replicated as needed. For $n >= 5$, synthesis time was longer than our timeout of 500 seconds, but it is reported for completeness. The graph is in logarithmic scale.

reasonable time (less than 500 seconds) for $n < 5$. Figure 7.5 illustrates a typical solution for

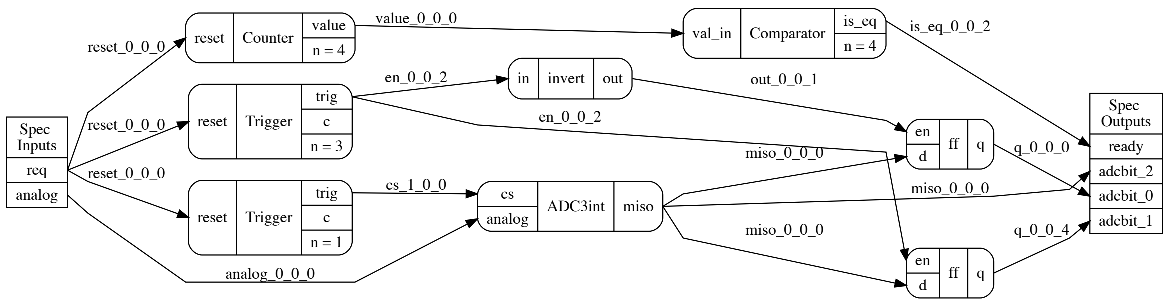


Figure 7.5: Typical synthesized composition for the ADC-SPI problem, in case of the specification and ADC with 3 bit resolution ($n = 3$).

$n = 3$.

7.8 Conclusion

In this chapter, we studied an approach to synthesis from libraries of LTL A/G contracts based on the CEGIS paradigm. Differently than the solution discussed in Chapter 6, here we use counterexamples generated by a verifier, i.e., a model checker, representing traces over the specification and system variables, to propose a more efficient solution. 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.

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. We applied our synthesis algorithm to three case studies, showing its effectiveness. In Chapter 8, we will build on top of the results discussed here to show how to further improve performance, by decomposing the system specification.

Chapter 8

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

In the previous chapters, we introduced techniques to perform synthesis of compositions of LTL A/G contracts (and components in general) that satisfy a certain specification. Synthesis tasks are feasible in many cases, but they are computationally intensive, especially for complex specifications. In this chapter, we describe an efficient technique to partition a specification, i.e., an LTL-based A/G 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.

The rest of the chapter is organized as follows. In Section 8.2, we reference the theoretical elements to support our work, and provide a formal analysis of our approach in Section 8.3. Then, in Section 8.5, we describe in detail our decomposition algorithm. In Section 8.6, 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 8.7.

8.1 Introduction

Given a specification described by an A/G contract, it is possible to satisfy it by composing a number of simpler contracts, where the theory of contracts provides the mathematical tools required to validate the design. When contracts are automatically chosen from a library of predefined designs, we talk about *Constrained Synthesis from Contract Libraries* (CSCL), Although synthesis tools and algorithms have come a long way, CSCL remains a computationally hard problem.

In this chapter, 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. 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.

This chapter builds on top of the concepts, problems, and solutions discussed in Chapter 7.

8.2 Preliminaries

In this chapter, we will refer to LTL A/G contracts and libraries as introduced in Chapter 3 and discussed further in Chapter 7, e.g., an LTL A/G contract is indicated as $C = (I, O, \varphi, \psi)$, and a library is a pair $L = (\mathcal{Z}, \mathcal{R})$.

As usual, we consider the system specification $S = (I_S, O_S, \varphi_S, \psi_S)$ that needs to be synthesized, as a LTL A/G contract.

The problem we want to solve is the same problem of synthesis from libraries of LTL A/G contracts introduced in Definition 10. In this chapter, we present a technique to improve the scalability of the synthesis process, which is reduced to a series of simpler tasks. In Section 8.4.1, we will formally describe such simpler synthesis tasks as a variation of the problem in Definition 10, that can still be solved, however, using the same techniques elaborated in Chapter 7. In this chapter, we leverage 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 $C_1 \otimes C_2 \preceq C$. Let also C_1', C_2' be contracts such that $C_1' \preceq C_1$, and $C_2' \preceq C_2$. Then, we have that $C_1' \otimes C_2' \preceq C$ holds, too.

8.3 Decomposing Contracts

Given a system specification expressed as a LTL A/G contract, our objective is to decompose it in several sub-specifications (or projections), to simplify the synthesis problem in Definition 10. 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 $C' = C_1' \otimes \cdots \otimes C_n'$ such that

$$C_1' \otimes \cdots \otimes C_n' \preceq C \tag{8.1}$$

Note that, compared to the problem addressed in Chapters 6 and 7, here each contract C_i' does not need to come from a library, but it will become itself the specification for another synthesis task. The independent implementability property of contracts, introduced in Section 3.2.1.4, guarantees that refining the single contracts on the left-hand side of Equation 8.1 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 we have discussed in Chapter 6, ultimately depends on the number of connections that the synthesizer needs to establish. Hence, the synthesis problem for each C'_i will need to be defined in a way that reduces the total number of connections that need to be found.

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, it should be also be found by synthesizing each C'_i. Thus, we would like contracts C'_is to be as "loose" as possible, meaning that their guarantees should be as weakest, and their assumptions the strongest, while still holding the refinement in Equation 8.1. Finally, the decomposition process should always be successful, meaning that, in the worst case, we should obtain a single contract such that $C'_1 \preceq C$.

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

8.3.1 Projection for LTL A/G Contracts

According to the discussion in the previous section, each contract resulting by 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. We call each C'_i of Equation 8.1 a *projection*. We do so to explicit our intent to have each C'_i define only some properties of the specification C, which is then fully realized only when all the projections are composed together.

Defining a projection for an A/G contract C, intuitively, means describing a new contract over a subset of its variables. The new assumption and guarantee will need, then, be defined only over those variables exposing, at the same time, all the behaviors that could be observed in C by looking at the subset of variables. Unfortunately, Wolper [84] 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. We provide a definition of projection for LTL A/G contracts that doesn't involve projecting LTL formulas, but it is able to partition only output variables of a certain contract. It is, however, sufficient for us to define the problem of contract decomposition, discussed in Section 8.3, and to provide a solution for it. The reasons behind the choice of defining the projection only over output variables will be clarified later.

Definition 11 (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*

$$\Pi_V(C) = C^c \theta_V \tag{8.2}$$

where C^c is a copy of C with fresh variables as in Definition 4, and θ_V specifies the following connections:

$$\forall p \in I : (C^c.x, x) \in \theta_V \tag{8.3}$$

$$\forall p \in O : p \in V \Leftrightarrow (C^c.x, x) \in \theta_V \tag{8.4}$$

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

$$\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C \tag{8.5}$$

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 10 (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

$$\Pi_X(C) = (\emptyset, \{a, a_0, b_0\}, \mathit{True}, \Box(a = a_0) \wedge \Box(a_0 \vee b_0))^1$$

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 $\Pi_X(C) \otimes \Pi_{\bar{X}}(C) \preceq C$ 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 guarantee were $\psi = \Box(a \wedge b)$, the set $X = \{a\}$ would yield a valid projection.

The notion of valid projection can also be used to justify the reason behind the decision of limiting the partition of variables to the set of output variables. Let us assume that the set V in Definition 11 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 10, there would be some variables of $\Pi_V(C)$ not mapped into C. Conversely, $\Pi_{\bar{V}}(C)$ would have all the input variables in V not mapped into C. Thus, their composition $C' = \Pi_V(C) \otimes \Pi_{\bar{V}}(C)$ would have $2 \cdot |I|$ input variables. The

[1]The new guarantee is computed according to the contract connection discussed in Section 3.2.1.6.

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 3.2.1.4, 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 $|I|$ new dummy variables to C, mapping them onto the unconnected variables of C'. The other problem, however, is substantial. Indeed, for each new input port, C' defines some constraints according to its assumption, 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 3.22c would not be satisfied as φ would allow more behaviors than φ', at least for any non-trivial assumption, i.e., different than $\varphi = \mathit{True}$.

Using the notion of projection for a LTL A/G contract we just introduced, we are now ready to formally introduce the problem we want to solve. Later, we will discuss how this problem and its solution address all the considerations we made at the beginning of Section 8.3.

Definition 12 (Contract Decomposition Problem (CD)). *Let $C = (I, O, \varphi, \psi)$ be a 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 $V_1, \dots, V_n$, with $n \geq 1$, such that*

$$\Pi_{V_1}(C) \otimes \Pi_{V_2}(C) \otimes \cdots \otimes \Pi_{V_n}(C) \preceq C \tag{8.6}$$

8.4 Solving 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 8.6 is satisfied.

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

Definition 13 (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 $\bar{V} = P \setminus V$:*

$$\forall \sigma : \sigma \not\models \varphi \Rightarrow \sigma_{Q\cup V} \not\models \varphi \vee \sigma_{Q\cup\bar{V}} \not\models \varphi \tag{8.7}$$

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

Example 11. *Let $\varphi = \Box(a \vee b)$, where $\Sigma = P = \{a, b\}$, and $Q = \emptyset$. Then $V = \{a\}$ does not contain independent variables*[2]*. Consider, for instance, the finite sequence $\sigma = [(a = False, b = False)]$. σ falsifies φ, but $\sigma_V = [(a = False)]$ does not, because the evaluation of b is unknown. On the other hand, for $\varphi' = \Box(a \wedge b)$ 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 8.5.

Theorem 8.4.1. *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. If variables in V are independent in O for ψ, then the projection $\Pi_V(C)$ is valid.*

Proof. 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 $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ must hold. This means verifying that:

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

We start from point 2). C is consistent, hence I-receptive, i.e., its guarantee 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 assumption of the composition $\Pi_V(C) \otimes \Pi_{\bar{V}}(C)$. 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. This proves the theorem. $\Box$

8.4.1 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 contract libraries.

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

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 8.6. The following theorem clarifies this point.

Theorem 8.4.2. *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*

$$\Pi_{V_1}(C) \otimes \cdots \otimes \Pi_{V_n}(C) \preceq C \tag{8.8}$$

Proof. By the definition of valid projection in Equation 8.5, we know that for each projection $\Pi_{V_i}(C)$ of a contract (I, O, φ, ψ), the refinement $\Pi_{V_i}(C) \otimes \Pi_{\bar{V_i}}(C) \preceq C$ holds, where $V_i \cup \bar{V_i} = O$ by construction. To prove Equation 8.8 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 8.4.1. 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 8.4.1. Hence, the theorem is proved. $\square$

Theorem 8.4.2 explains how we can partition a contract C using n valid projections, $\Pi_{V_1}(C), \ldots, \Pi_{V_n}(C)$. 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 $\Pi_{V_1}(C), \ldots, \Pi_{V_n}(C)$. That is, if there exist compositions such that

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

for some $m_1, \ldots, m_n$, then by the *independent development* property in [10], the following holds:

$$\left.\begin{array}{c} C_1^{V1} \otimes \cdots \otimes C_{m1}^{V1} \\ \otimes \\ C_1^{V2} \otimes \cdots \otimes C_{m2}^{V2} \\ \otimes \\ \vdots \\ \otimes \\ C_1^{Vn} \otimes \cdots \otimes C_{mn}^{Vn} \end{array}\right\} \preceq \left.\begin{array}{c} \Pi_{V1}(C) \\ \otimes \\ \Pi_{V2}(C) \\ \otimes \\ \vdots \\ \otimes \\ \Pi_{Vn}(C) \end{array}\right\} \preceq C \tag{8.9}$$

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 solve this issue by defining a variant of the CSCL problem in Definition 10.

Definition 14 (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:*

$$(C_1 \otimes \cdots \otimes C_m)\theta \otimes \Pi_{\bar{V}}(S) \preceq S \tag{8.10}$$

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 Chapter 7.

Once we have found a solution for all the projections, Equation 8.9 guarantees that putting all the pieces back together will result in a proper refinement of our original system specification.

By only synthesizing the variables in V, moreover, 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 $\Pi_V(C) \otimes \Pi_{\bar{V}}(C)$ 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 $\Pi_V(C)$.

8.5 An Efficient Decomposition Algorithm

In this section we discuss an efficient algorithm to decompose a contract $C = (I, O, \varphi, \psi)$ following the concepts discussed in the previous sections. One obvious possibility would be to exhaustively check whether $\Pi_{V_i}(C) \otimes \Pi_{\bar{V}_i}(C) \preceq C$ 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 8.5 at least $2^{|O|}$ 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 until it contains independent variables. We do so by analyzing the counterexamples obtained verifying some *ad hoc* formulas.

```
1  function DecomposeContract:
       Input: Contract $C = (I, O, \varphi, \psi)$
       Output: Set of valid contract projections
2      clusters $\leftarrow \{\}$;
3      for $p \in O$ do
4          $V \leftarrow \{p\}$;
5          passed $\leftarrow$ $False$;
6          repeat
7              $\bar{V} \leftarrow O \setminus V$;
8              passed, trace $\leftarrow$ checkValid($\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$);
9              if $not$ passed then
10                 $D \leftarrow$ ParseTrace(trace);
11                 $V \leftarrow V \cup D$;
12             end
13         until passed;
14         clusters $\leftarrow$ clusters $\cup V$;
15     end
16     clusters $\leftarrow$ MergeClusters(clusters);
17     return $\{\Pi_V(C) \mid V \in$ clusters $\}$
18 end
```

Algorithm 12: Contract Decomposition algorithm. It takes a contract C as input, and returns a set of n projections such that $\Pi_{V_1}(C) \otimes \cdots \otimes \Pi_{V_n}(C) \preceq C$.

Algorithm 12 describes the main decomposition algorithm. For each output variable p (Line 3), we start with a set V containing only p (Line 4). After creating the candidate projections $\Pi_V(C)$ and $\Pi_{\bar{V}}(C)$, the algorithm verifies that $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ in Line 8 (by model checking the refinement formulas). If $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ holds, then passed will be true, meaning that $\Pi_V(C)$ is a valid projection, and we can move on to the next iteration (Line 14). If, however, $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ 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, to add them to V (Lines 10 and 11), and repeat the process until $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ 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 12. The algorithm always terminates, as in the worst case we have that $V = O$, thus $\Pi_V(C) = C$ and $\bar{V} = \emptyset$, which always verifies $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$. It does so 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.

8.5.1 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 $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ is true, which means that the projection over V is valid. We say so because, otherwise, the model-checker would have found a trace in which C is false, but $\Pi_V(C) \otimes \Pi_{\bar{V}}(C)$ is not, according to Definition 13. If the refinement does not hold, 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 $\bar{V}$. 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 it to V. This process stops when $\Pi_V(C) \otimes \Pi_{\bar{V}}(C) \preceq C$ is true, which means, as we have seen above, that V contains independent variables. In the worst case, the process stops when $V = O$, which will always result in a valid projection. The algorithm is therefore sound.

We cannot claim that the algorithm is complete, meaning that if there exists a partition $V_1, \dots, V_n$ such that $\Pi_{V_1}(C) \otimes \cdots \otimes \Pi_{V_n}(C) \preceq C$, with $n > 1$, we cannot guarantee that the algorithm will find it. The reason is that 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.

8.6 Evaluation

We implemented the proposed algorithm in a modified version of PYCO, the tool described in Chapter 7. All the experiments were run on a 3.3 GHz Intel Xeon machine, with 32GB of RAM, limiting the maximum number of parallel processes to 8.

8.6.1 The EPS problem

The first problem we used to test our synthesis with decomposition approach was the EPS problem described in Chapter 4. As discussed also in previous chapters,

our goal is to synthesize the logic of the BCPU from a set of subsystem controllers, described by the library of 20 LTL A/G contracts illustrated in Table 4.2. The system specification, instead, is represented in Table 4.1.

We ran 9 synthesis tasks with increasing complexity (each task was an incremental subset of the guarantees of Table 4.1), for two libraries with 20 and 40 contracts. Figure 8.1 illustrates the results we obtained. A typical solution satisfying all the specifications consisted

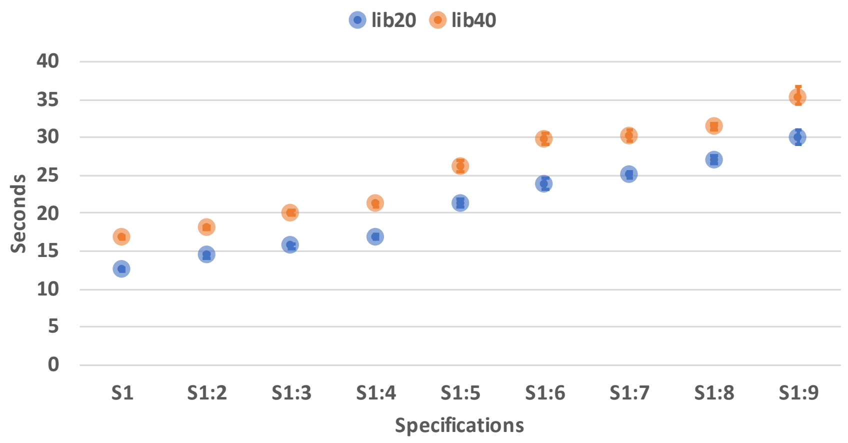


Figure 8.1: Summary of the EPS experiments. We synthesized a controller using two different libraries, one with 20 and the other with 40 contracts, defined in Table 4.2. For each category, we ran 30 experiments and then bootstrapped the sample. Each point represents the bootstrap mean synthesis time, while the bars represent its 95% bootstrap confidence interval.

of 6 components, similar to the composition in Figure 6.9. 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 $V_7 = \{C_{10}\}$.

When compared to the results in Section 7.7.2, the synthesis approach described in this chapter did present a comparable performance. This is not surprising, at least for this example, because the technique discussed in Section 7.7.2 is already very efficient in terms of number of queries to the underlying model checker. In this case, however, we need to consider that the decomposition algorithm is executed before synthesis before each experiment, without impacting significantly on the overall synthesis performance.

8.6.2 The SPI-ADC problem

A second problem is the SPI-ADC example introduced in Section 7.7.3. Here, our goal is to generate an SPI-like controller for an ADC. This problem is more challenging than the previous one, as synthesis was unfeasible already for a specification requiring 5-bit resolution from the ADC. In this section, we compare those results to the ones obtained by synthesizing the same controllers using the decomposition technique discussed earlier.

Figure 8.2 summarize the results of our experiments, which represent the typical performance for each configuration. The benefit of decomposing the specification can be observed immediately. In this case, in fact, we were able to synthesize specifications requiring up to 8-bit resolution, in considerably less time. Figure 8.3 shows the typical solution for a specification requiring an ADC with 8 bit resolution. As one can observe, the synthesizer correctly inferred the parameters for several components (e.g., the triggers), to make sure

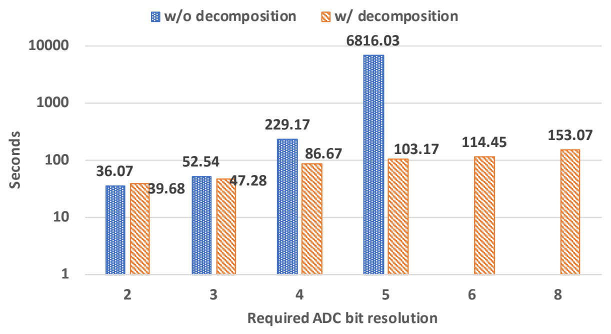


Figure 8.2: Summary of the results for the ADC-SPI experiments, for specifications with parameter $n \in \{2, 3, 4, 5, 6, 8\}$ and library size of 11 contracts, respectively. Each library had the appropriate ADC according to n. The other contracts, including the parametric ones, were replicated as needed. The graph is in logarithmic scale.

that the rest of the signals had the correct timing.

8.7 Conclusion

In this chapter, we presented a technique to increase the 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 tested this decomposition approach to the synthesis problems analyzed in Chapter 7. 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.

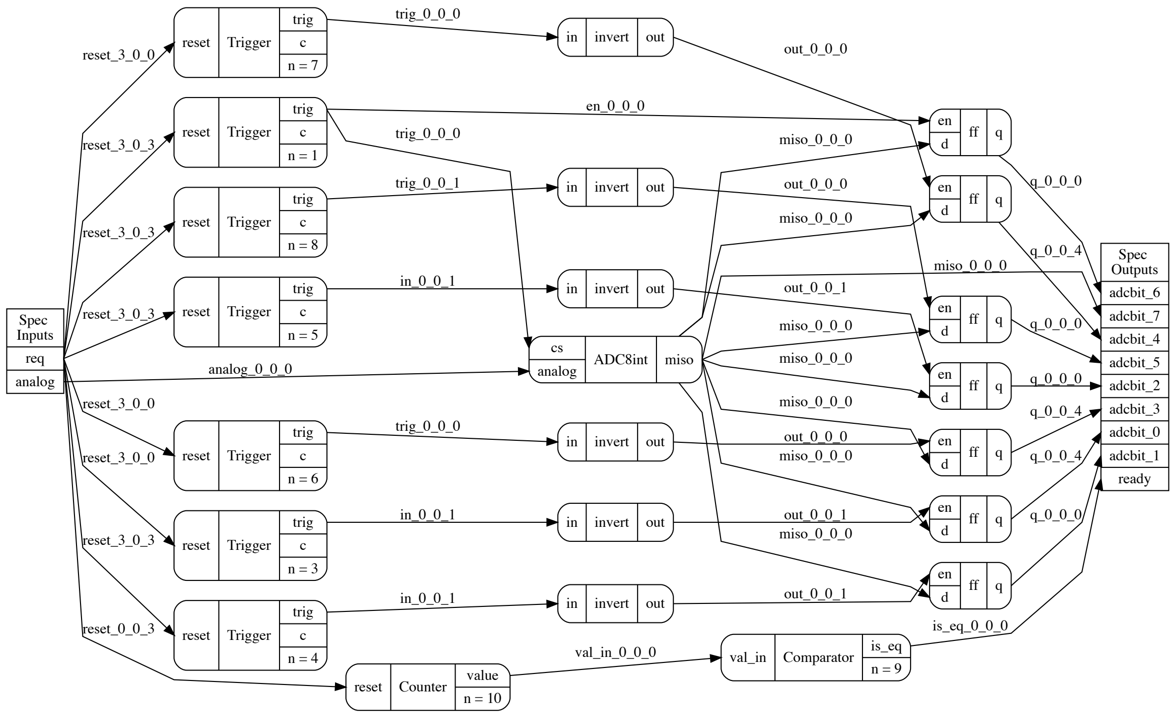


Figure 8.3: Typical synthesized composition for the ADC-SPI problem, in case of the specification and ADC with 8-bit resolution ($n = 8$).

Chapter 9

Conclusions

The work described in this dissertation is centered on two fundamental aspects of system design, which are the problem of verifying the correctness of user-provided designs, and the problem of synthesizing a design by composing components available in a library. In both cases, we focused on specifications and components described as LTL A/G contracts.

We started by introducing the A/G contract framework that we used as the underlying formalism for the rest of the work, showing how assumptions and guarantees can be represented using LTL formulas. In particular, we described the explicit definition of connections between contracts, which is used to characterize compositions in both the verification and synthesis problems. We also discussed a technique to check contract refinement by framing the problem as a model checking task, and showed how to use off-the-shelf tools, such as the NuXMV model checker, to execute such task.

We addressed the problem of improving the refinement checking performance by using pre-computed contract relations stored in the contract library. In this case, the library contains both regular contracts and several subsystems realized as a composition of the former, specifying explicitly the refinement relations between the two. This information is used to speed up the verification of refinement between a system specification and a composition of library contracts, provided by the user, describing the system. The verification engine, then, iteratively builds abstractions by leveraging the relations in the library and tries to verify refinement, at each step, on one of those simplified designs. Proving the refinement using a system abstraction is more efficient, as it allows to derive smaller model checking problems. We evaluated this approach by verifying properties on a design for an aircraft EPS controller.

Then, we focused on the problem of synthesis of contract compositions that need to satisfy a system specification, providing several solutions based on the OGIS paradigm. In OGIS, a solution is identified through a collaboration between a synthesizer, which provides candidates, and a verifier, which checks whether those candidates are correct. First, we described a synthesis technique based on the assumption that the only output of the verifier is a yes/no answer on the correctness of a candidate. This assumption allowed us to define a strategy that is more general, and can work also with formalisms other than LTL A/G contracts. Then, we removed the restriction on the information available after the verification step, assuming that a verifier would be able to return richer counterexamples, i.e., traces over the system and specification variables, showing why a candidate is not correct. The second synthesis strategy we presented uses these counterexamples to build partial models of the environment to drive the synthesis process. The resulting algorithm is more efficient than the previous one in terms of number of calls to the verifier, although the process of generating candidates is more onerous.

We improved further the synthesis performance by introducing a technique to decompose a synthesis problem, for an LTL A/G contract representing a system specification, into several smaller, independent tasks. We did so by defining a projection operation for contracts that can be used to identify groups of independent variables. Each synthesis tasks, then, can focus on solving the problem for that smaller set of variables, where there is a formal guarantee, under some assumptions, that composing back all the partial solutions refines the original specification. Moreover, the search space is guaranteed to be the same, meaning that any solution that can be found with a centralized synthesis approach can be also found solving the decomposed specification.

We evaluated all the synthesis techniques on several case studies, including the EPS problem. The experiments on the case studies provided valuable insights on the synthesis performance, and showed the capabilities of our approach.

Future Directions We believe that the techniques proposed in this dissertation will play a central role in the realization of more general platform-based design methodologies and tools handling LTL specifications. We identify several future directions based on our results.

The synthesis technique described in Chapter 7 is efficient, but we don't have, yet, a way to specify sophisticated heuristics to help the synthesizer converge faster. In Chapter 6 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 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 [46], for instance, can be used to represent "what is missing" in a certain composition to satisfy the specification. Additionally, a recent work [45] 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.

Another interesting extension of the work described in Chapter 7 is the definition of hierarchical synthesis techniques based on CEGIS. In this case, a library would be organized similarly to the libraries in Chapter 5, which include refinement relations between the contained contracts and their composition. The synthesis algorithm, then, would try to build a design in an orderly fashion, starting from the hierarchical level including the most abstract contracts and iteratively going all the way down to the most concrete ones.

The notion of specification decomposition described in Chapter 8 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 that has smaller sets of independent variables. This could translate in a tool able to suggest to a designer how to "tweak" a certain specification to maximize synthesis performance.

Bibliography

[1] Luca de Alfaro and Thomas 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. Vienna, Austria: ACM, 2001, pp. 109–120. ISBN: 1-58113-390-1. DOI: `10.1145/503209.503226`.

[2] Luca de Alfaro and Thomas A. Henzinger. “Interface Theories for Component-Based Design”. In: *Embedded Software*. Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 148–165. ISBN: 978-3-540-45449-6.

[3] Rajeev Alur, Milo M. K. Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. “Synthesizing Finite-State Protocols from Scenarios and Requirements”. In: *Haifa Verification Conference*. Vol. 8855. Lecture Notes in Computer Science. Springer, 2014, pp. 75–91.

[4] Rajeev Alur, Salar Moarref, and Ufuk Topcu. “Compositional Synthesis with Parametric Reactive Controllers”. In: *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. HSCC ’16. Vienna, Austria: ACM, 2016, pp. 215–224. ISBN: 978-1-4503-3955-1. DOI: `10.1145/2883817.2883842`.

[5] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. “Automatic Completion of Distributed Protocols with Symmetry”. In: *CAV (2)*. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 395–412.

[6] Rajeev Alur and Stavros Tripakis. “Automatic Synthesis of Distributed Protocols”. In: *SIGACT News* 48.1 (Mar. 2017), pp. 55–90. ISSN: 0163-5700. DOI: `10.1145/3061640.3061652`.

[7] A. Armando, R. Carbone, and L. Compagna. “LTL Model Checking for Security Protocols”. In: *20th IEEE Computer Security Foundations Symposium (CSF’07)*. July 2007, pp. 385–396. DOI: `10.1109/CSF.2007.24`.

[8] László Babai and Eugene M. Luks. “Canonical labeling of graphs”. In: *Proc. of ACM Symp. on Theory of Computing*. STOC ’83. 1983, pp. 171–183. ISBN: 0-89791-099-0.

[9] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability modulo theories". English (US). In: *Handbook of Satisfiability.* 1st ed. Vol. 185. Frontiers in Artificial Intelligence and Applications 1. 2009, pp. 825–885. ISBN: 9781586039295. DOI: `10.3233/978-1-58603-929-5-825`.

[10] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. "Multiple Viewpoint Contract-Based Specification and Design". In: *Formal Methods for Components and Objects.* Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 200–225. ISBN: 978-3-540-92188-2.

[11] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. "Contracts for System Design". In: *Foundations and Trends in Electronic Design Automation* 12.2-3 (2018), pp. 124–400. ISSN: 1551-3939. DOI: `10.1561/1000000053`.

[12] A. Bhatia, L. E. Kavraki, and M. Y. Vardi. "Sampling-based motion planning with temporal goals". In: *2010 IEEE International Conference on Robotics and Automation.* May 2010, pp. 2689–2696. DOI: `10.1109/ROBOT.2010.5509503`.

[13] Armin Biere. "Bounded Model Checking". In: *Handbook of Satisfiability.* Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 457–481.

[14] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. "RATSY – A New Requirements Analysis Tool with Synthesis". In: *Computer Aided Verification.* Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 425–429. ISBN: 978-3-642-14295-6.

[15] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Trans. Comput.* 35.8 (Aug. 1986), pp. 677–691. ISSN: 0018-9340. DOI: `10.1109/TC.1986.1676819`.

[16] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. "The nuXmv Symbolic Model Checker". In: *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559.* New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 334–342. ISBN: 978-3-319-08866-2. DOI: `10.1007/978-3-319-08867-9_22`.

[17] A. Cimatti and S. Tonetta. "A Property-Based Proof System for Contract-Based Design". In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications.* Sept. 2012, pp. 21–28. DOI: `10.1109/SEAA.2012.68`.

[18] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. “NuSMV 2: An OpenSource Tool for Symbolic Model Checking”. In: *Proceedings of the 14th International Conference on Computer Aided Verification.* 2002.

[19] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-Guided Abstraction Refinement”. English. In: *Computer Aided Verification.* Ed. by E.Allen Emerson and AravindaPrasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-67770-3. DOI: 10.1007/10722167_15.

[21] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. “Learning Assumptions for Compositional Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* 2003, pp. 331–346.

[22] Eric Dallal and Paulo Tabuada. “Decomposing controller synthesis for safety specifications”. In: *2016 IEEE 55th Conference on Decision and Control (CDC).* Dec. 2016, pp. 5720–5725. DOI: 10.1109/CDC.2016.7799148.

[23] Bruce Powel Douglass. *Agile Systems Engineering.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 9780128023495.

[24] Laurent Doyen, Thomas A. Henzinger, Barbara Jobstmann, and Tatjana Petrov. “Interface Theories with Component Reuse”. In: *Proceedings of the 8th ACM International Conference on Embedded Software.* EMSOFT ’08. Atlanta, GA, USA: ACM, 2008, pp. 79–88. ISBN: 978-1-60558-468-3. DOI: 10.1145/1450058.1450070.

[25] Georgios E. Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J. Pappas. “Temporal Logic Motion Planning for Dynamic Robots”. In: *Automatica* 45.2 (Feb. 2009), pp. 343–352. ISSN: 0005-1098. DOI: 10.1016/j.automatica.2008.08.008.

[26] Emmanuel Filiot, Naiyong Jin, and Jean-Franccois Raskin. “An Antichain Algorithm for LTL Realizability”. In: *Computer Aided Verification.* Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 263–277. ISBN: 978-3-642-02658-4.

[27] Emmanuel Filiot, Naiyong Jin, and Jean-Franccois Raskin. “Antichains and compositional algorithms for LTL synthesis”. In: *Formal Methods in System Design* 39.3 (Dec. 2011), pp. 261–296. ISSN: 1572-8102. DOI: 10.1007/s10703-011-0115-3.

[28] I. Filippidis, S. Dathathri, S. C. Livingston, N. Ozay, and R. M. Murray. “Control design for hybrid systems with TuLiP: The Temporal Logic Planning toolbox”. In: *2016 IEEE Conference on Control Applications (CCA).* Sept. 2016, pp. 1030–1041. DOI: 10.1109/CCA.2016.7587949.

[29] I. Filippidis and R. M. Murray. “Symbolic construction of GR(1) contracts for systems with full information”. In: *2016 American Control Conference (ACC)*. July 2016, pp. 782–789. DOI: 10.1109/ACC.2016.7525009.

[30] Ioannis Filippidis. “Decomposing Formal Specifications Into Assume-Guarantee Contracts for Hierarchical System Design”. PhD thesis. California Institute of Technology, July 2018.

[31] Robert W. Floyd. “Assigning Meanings to Programs”. In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32.

[32] Paul Gastin and Denis Oddoux. “Fast LTL to Büchi Automata Translation”. In: *Proceedings of the 13th International Conference on Computer Aided Verification (CAV’01)*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Paris, France: Springer, July 2001, pp. 53–65.

[33] Orna Grumberg and David E. Long. “Model Checking and Modular Verification”. In: *ACM Transactions on Programming Languages and Systems* 16 (1991).

[34] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. “Synthesis of Loop-free Programs”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 62–73. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993506.

[35] Anubhav Gupta, K.L. McMillan, and Zhaohui Fu. “Automated assumption generation for compositional verification”. In: *Formal Methods in System Design* (2008), pp. 285–301.

[36] Duane Hanselman. *Brushless permanent magnet motor design*. Cranston, R.I, USA: The Writers’ Collective, 2003. ISBN: 9781932133639.

[37] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. “Decomposing Refinement Proofs Using Assume-guarantee Reasoning”. In: *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*. ICCAD ’00. San Jose, California: IEEE Press, 2000, pp. 245–253. ISBN: 0-7803-6448-1.

[38] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. “You Assume, We Guarantee: Methodology and Case Studies”. In: *Proceedings of the 10th International Conference on Computer Aided Verification*. CAV ’98. London, UK, UK: Springer-Verlag, 1998, pp. 440–451. ISBN: 3-540-64608-6. URL: http://dl.acm.org/citation.cfm?id=647767.733780.

[39] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[40] Gerard J. Holzmann. “Explicit-State Model Checking”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Cham: Springer International Publishing, 2018, pp. 153–171. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_5.

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

[42] A. Iannopollo, S. Tripakis, and A. Sangiovanni-Vincentelli. "Specification decomposition for synthesis from libraries of LTL Assume/Guarantee contracts". In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Mar. 2018, pp. 1574–1579. DOI: `10.23919/DATE.2018.8342266`.

[43] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained Synthesis from Component Libraries". In: *Formal Aspects of Component Software: 13th International Conference, FACS 2016, Besanccon, France, October 19-21, 2016, Revised Selected Papers*. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Springer International Publishing, 2017, pp. 92–110. ISBN: 978-3-319-57666-4. DOI: `10.1007/978-3-319-57666-4_7`.

[44] Antonio Iannopollo, Stavros Tripakis, and Alberto Sangiovanni-Vincentelli. "Constrained synthesis from component libraries". In: *Science of Computer Programming* 171 (2019), pp. 21–41. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2018.10.003`.

[45] Íñigo Íncer Romeo, Marten Lohstroh, Antonio Iannopollo, Edward A. Lee, and Alberto Sangiovanni- Vincentelli. "A Metric for Linear Temporal Logic". In: *arXiv e-prints*, arXiv:1812.03923 (Nov. 2018), arXiv:1812.03923. arXiv: `1812.03923 [cs.LO]`.

[46] Íñigo Íncer Romeo, Alberto Sangiovanni-Vincentelli, Chung-Wei Lin, and Eunsuk Kang. "Quotient for Assume-Guarantee Contracts". In: *Conference on Formal Methods and Models for System Design (MEMOCODE), 16th ACM/IEEE International Conference on*. 2018.

[47] Texas Instruments. *KeyStone Architecture - Serial Peripheral Interface (SPI)*. Mar. 2012. URL: `https://web.archive.org/web/20180328180445/http://www.ti.com:80/lit/ug/sprugp2a/sprugp2a.pdf`.

[48] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. "Oracle-Guided Component-Based Program Synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. May 2010, pp. 215–224.

[49] Susmit Jha and Sanjit A. Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54.7 (Nov. 2017), pp. 693–726. ISSN: 1432-0525. DOI: `10.1007/s00236-017-0294-5`.

[50] Susmit Jha and Sanjit A. Seshia. "Are There Good Mistakes? A Theoretical Analysis of CEGIS". In: *3rd Workshop on Synthesis (SYNT)*. July 2014, pp. 84–99.

[51] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[52] Leslie Lamport. “What Good Is Temporal Logic?” In: *Information Processing 83, R. E. A. Mason, ed., Elsevier Publishers* 83 (May 1983), pp. 657–668.

[53] E. A. Lee and A. Sangiovanni-Vincentelli. “A framework for comparing models of computation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 17.12 (Dec. 1998), pp. 1217–1229. ISSN: 0278-0070. DOI: `10.1109/43.736561`.

[54] Yoad Lustig and Moshe 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. York, UK: Springer-Verlag, 2009, pp. 395–409. ISBN: 978-3-642-00595-4. DOI: `10.1007/978-3-642-00596-1_28`.

[55] Panagiotis Manolios, Jorge Pais, and Vasilis Papavasileiou. “The Inez Mathematical Programming Modulo Theories Framework”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 53–69.

[56] K. L. McMillan. *The SMV Language*. Tech. rep. Cadence Berkeley Labs, Mar. 1999.

[57] Bertrand Meyer. “Applying ”Design by Contract””. In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162. DOI: `10.1109/2.161279`.

[58] Rodney G. Michalko. “Electrical starting, generation, conversion and distribution system architecture for a more electric vehicle”. Patent US20060061213A1 (US). Honeywell International Inc. Mar. 2006.

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

[60] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[61] P. Nuzzo, J.B. Finn, A. Iannopollo, and A.L. Sangiovanni-Vincentelli. “Contract-based design of control protocols for safety-critical cyber-physical systems”. In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. Mar. 2014, pp. 1–4. DOI: `10.7873/DATE.2014.072`.

[62] P. Nuzzo, A. Iannopollo, S. Tripakis, and 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*. Oct. 2014, pp. 104–113. DOI: `10.1109/MEMCOD.2014.6961848`.

[63] N. Ozay, U. Topcu, and R. M. Murray. “Distributed power allocation for vehicle management systems”. In: *2011 50th IEEE Conference on Decision and Control and European Control Conference*. Dec. 2011, pp. 4841–4848. DOI: `10.1109/CDC.2011.6161470`.

[64] Alessandro Pinto and Alberto L. Sangiovanni Vincentelli. “CSL4P: A Contract Specification Language for Platforms”. In: *Systems Engineering* 20.3 (2017), pp. 220–234. DOI: `10.1002/sys.21386`.

[65] Nir Piterman, Amir Pnueli, and Yaniv Sa’ar. “Synthesis of Reactive(1) Designs”. In: *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2006.

[66] A. Pnueli and R. Rosner. “Distributed reactive systems are hard to synthesize”. In: *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*. Oct. 1990, 746–757 vol.2. DOI: `10.1109/FSCS.1990.89597`.

[67] A. Pnueli and 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. Austin, Texas, USA: ACM, 1989, pp. 179–190. ISBN: 0-89791-294-2. DOI: `10.1145/75277.75293`.

[68] Amir Pnueli. “The Temporal Logic of Programs”. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[69] P. J. Ramadge and W. M. Wonham. “Supervisory Control of a Class of Discrete Event Processes”. In: *SIAM Journal on Control and Optimization* 25.1 (1987), pp. 206–230. DOI: `10.1137/0325013`.

[70] Rohit Ramesh, Richard Lin, Antonio Iannopollo, Alberto Sangiovanni-Vincentelli, Björn Hartmann, and Prabal Dutta. “Turning Coders into Makers: The Promise of Embedded Design Generation”. In: *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*. SCF ’17. Cambridge, Massachusetts: ACM, 2017, 4:1–4:10. ISBN: 978-1-4503-4999-4. DOI: `10.1145/3083157.3083159`.

[71] Roni Rosner. “Modular synthesis of reactive systems”. PhD thesis. Weizmann Institute of Science, 1992.

[72] A. Sangiovanni-Vincentelli. “Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design”. In: *Proceedings of the IEEE* 95.3 (Mar. 2007), pp. 467–506. ISSN: 0018-9219. DOI: `10.1109/JPROC.2006.890107`.

[73] A. Sangiovanni-Vincentelli and G. Martin. “Platform-based design and software design methodology for embedded systems”. In: *IEEE Design Test of Computers* 18.6 (Nov. 2001), pp. 23–33. ISSN: 0740-7475. DOI: `10.1109/54.970421`.

[74] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. “Benefits and Challenges for Platform-based Design”. In: *Proceedings of the 41st Annual Design Automation Conference*. DAC ’04. San Diego, CA, USA: ACM, 2004, pp. 409–414. ISBN: 1-58113-828-8. DOI: `10.1145/996566.996684`.

[75] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. “Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems”. In: *European Journal of Control* 18.3 (June 2012), pp. 217–238.

[76] S. A. Seshia. “Combining Induction, Deduction, and Structure for Verification and Synthesis”. In: *Proceedings of the IEEE* 103.11 (Nov. 2015), pp. 2036–2051. ISSN: 0018-9219. DOI: `10.1109/JPROC.2015.2471838`.

[77] Marco Sgroi. “Platform-based Design methodologies for Communication Networks”. PhD thesis. University of California, Berkeley, 2002.

[78] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press, 1983. ISBN: 0262192187.

[79] M Sinnett. “787 No-Bleed Systems: Saving Fuel and Enhancing Operational Efficiencies”. In: *AEROMAGAZINE* 7 (Jan. 2007), pp. 6–11.

[80] A. P. Sistla and E. M. Clarke. “The Complexity of Propositional Linear Temporal Logics”. In: *J. ACM* 32.3 (July 1985), pp. 733–749. ISSN: 0004-5411. DOI: `10.1145/3828.3837`.

[81] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. “Combinatorial Sketching for Finite Programs”. In: ASPLOS XII (2006), pp. 404–415. DOI: `10.1145/1168857.1168907`.

[82] J. Torán. “On the Hardness of Graph Isomorphism”. In: *SIAM Journal on Computing* 33.5 (2004), pp. 1093–1108.

[83] David D. Walden, Garry J. Roedler, Kevin Forsberg, R. Douglas Hamelin, and Thomas M. Shortell, eds. *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*. 4th ed. Hoboken, NJ: Wiley, 2015. ISBN: 978-1-118-99940-0.

[84] P. Wolper. “Temporal logic can be more expressive”. In: *Foundations of Computer Science*. 1981, pp. 340–348.

[85] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. “Reasoning About Infinite Computation Paths”. In: *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*. SFCS ’83. Washington, DC, USA: IEEE Computer Society, 1983, pp. 185–194. ISBN: 0-8186-0508-1. DOI: `10.1109/SFCS.1983.51`.

[86] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard Murray. *Automatic Synthesis of Robust Embedded Control Software*. 2010.